目錄

介绍 1.1
第一部分 1.2
    前序 1.2.1
    序 1.2.2
    C代码风格 1.2.3
    C代码规范 1.2.4
    变量 1.2.5
第二部分 1.3
    指针(上) 1.3.1
    指针(下) 1.3.2
    预处理器 1.3.3
    效率至上(上) 1.3.4
    效率至上(下) 1.3.5
    未曾领略的新风景 1.3.6
    C线程和Glib的视角 1.3.7
第三部分 1.4
    错误处理 1.4.1
    开始行动 1.4.2
    单线程备份(上) 1.4.3
    单线程备份(下) 1.4.4
    多线程备份 1.4.5
    总结 1.4.6
第四部分 1.5
    网络的世界 1.5.1
    套接字编程(1) 1.5.2
    套接字编程(2) 1.5.3
    并发HTTP服务器(1) 1.5.4

并发HTTP服务器(2) 1.5.5

并发HTTP服务器(3) 1.5.6

并发HTTP服务器(4) 1.5.7

并发HTTP服务器(5) 1.5.8

C 语言进阶

> 没有那么多条条框框
>
> 总之看了总会有收获
>
> 需要离线版本可以前往 Gitbook 页面下载，链接在下方

适用人群

对于已经掌握 C 语言基础的人，本书是非常值得学习参考的，作者打破传统 C 语言学习的繁琐过程，个性展示不一样的 C。

版权信息

作者 Wrestle.Wu 授权极客学院转载，如需转载，请标明出处。

> 联系方式：https://github.com/wrestle
>
> 项目地址：https://github.com/wrestle/gitbook

第一部分

> **C**语言核心知识 上

我的C语言

0x00-C语言前续工作

正所谓，工欲善其事，必先利其器，把握住当下最强大的工具，能让我们在学习的道路上少走许多弯路，多吸取前人的失败经验，能让自己快速成长，因为成长总是在消耗我们的耐心以及生命。

入门或者精通或者应用，不管哪一方面，对于一个编程语言而言，最方便的还是使用一个**IDE**作为你的有力助手，什么事**IDE**？通俗而专业的说叫做集成开发环境，这个通过字面就能理解到了，就是所有其他的事情都不需要程序员操心，你需要操心的就是写出代码，至于代码完成之后的一系列工作，都不需要你来管，IDE一键帮你搞定。

当然，会有许多前辈告诉你，如果你想理解C语言，那你一定要使用最基层的东西来写，比如XXX编辑器配上XXX编译器，晕头转向之后更加茫然，本来就支离破碎的小心肝，又被粉碎了一次，撇开那些与当下不符合的幻想，活在现实中，选择一个适合你的**IDE**，逐渐适应它。

- 讲几个著名的**IDE**，并给出建议，利器第一步：

    1. 宇宙级的**IDE**: Visual Studio(2010~2017)，之所以说宇宙级，因为这是市面上最强大的集成开发环境，由微软公司出品，但是放在开头不是为了推荐他，而是为了警示大家不要使用它作为C语言的集成开发环境，因为它使用的是微软公司自己定制的C++编译器，也就是说，你的C语言代码会在C++的标准下编译运行，这就是一个十分不好的现象，即便是C++我依旧不喜欢使用Visual Studio，因为它的C++编译器总是和普通的标准有所出入。

        原归正传，Visual Studio的确不是一个好的C语言学习IDE，所以请另外选择一个。

    2. 老牌**IDE**: DevC++，这又是一个大家耳熟能详，经常能在老师手里看见的C语言教学利器，但是，它是C++的IDE，记住C于C++完全是两个世界的人，虽然C++宣称能全面兼容C程序，但是有些东西依旧是有所区别，体现在语法的兼容性上，后文会有提及。那为什么大学老师喜欢使用它呢？因为一本由清华大学出版社出版的《数据结构》，让无数人为之折服，其

中赫然写着由于性能我们不能拘泥于小细节，故对于C++的特性 `&` 引用，我们可以将其使用在C语言的语法中，就是这句话，让无数无知的学子扑向其中，再也分不清C与C++，看成谭浩强之后的，清华大学出版社又一诲人不倦的力作。

所以，真爱编程，远离清华大学出版社，也请大家注意，不要使用DevC++这个IDE进行C语言程序的开发以及练习。

3. 知名**IDE**: Code::Blocks，是一款非常优秀的开源跨平台集成开发环境，体积并不大，适合作为C语言的IDE，并且功能齐全，有兴趣的人可以深究，这是几个首推的C语言开发环境选择。

4. 知名**IDE**: CodeLite，是一款非常优秀的开源跨平台集成开发环境，体积并不大，适合作为C语言的IDE，并且功能齐全，有兴趣的人可以深究，这是次推的C语言开发环境选择，因为使用起来稍微也有些额外的工作要做。

5. 著名**IDE**: Xcode，是一个苹果电脑上的史诗级集成开发环境，虽然脱胎于C语言，但是由于某些原因，并不太建议使用其作为C语言的开发环境。

6. 实用的 **IDE**: CLion, 是一款收费软件，但是对于学生教师免费，你只需要使用教育邮箱进行一些验证步骤就能得到它，有条件的推荐这个IDE，缺点就是使用 `Java` 编写，实在是有些慢。

7. 知名的 **IDE**: Qt Creator 是一款免费的集成开发环境，跨平台，且有远程调试功能，十分推荐使用！但是初学者可能对项目工程没什么概念，会比较麻烦。

8. 不知名 **IDE**: Kdevelop5 尚处于测试阶段，容易崩溃，但却是一款开源的集成开发环境！等稳定下来说不定又是一把利器。

- IDE的基本配置 利器第二步是对所选的IDE进行一些基本的配置，以及小科普。

  1. 对于一个练习C语言的开发环境来说，选择合适的标准和编译器是很重要的，在Windows以及Linux操作系统下，我们还是使用 `GCC` 这个家伙比较多，开源，免费，且极其强大。当然你也可以选择 `clang` ，当然整个计算机领域中支持C语言的编译器并不止这两个，只不过这两个是开源免费，而且功能强大，十分适合作为个人开发以及无特殊需求的企业开发的选择。

     选择了编译器，我们开始讲标准：

对于 `GCC 5.1` 以下的所有版本，都默认对C语言使用 `C89` 标准，但是我建议使用 `C99` 两者的差距，有一个极其明显的地方，便是 `for` 循环的使用

```
/*C89:*/
int i;
for(i = 0;i < 10;++i)
/*...*/
```

...

```
/*C99*/
for(int i = 0;i < 10;++i)
/*...*/
```

这只是其中的一种差别，但是C99需要人为手动的开启，但是很多人有疑问，为什么有时候没有配置什么也能使用后面的语法？吴老师告诉你，这是因为你用了C++的文件进行C语言的开发，就像挂羊皮卖狗肉的道理。

2. 开启C99

   一般IDE的顶部都是一系列的标签，找到 `工具/设置` ，因为不同的IDE可能有不同的标签，总之在其中找到一个叫**(**编译器**)Compiler**之后，在其中的**other option**中加入以下: `-std=c99` ，这便是开启C99的选项代码，完事之后保存即可。虽然说我们是中国人，但是毕竟这东西的外国人发明的，我们能看英文就看英文吧。

3. 至此，利器成功配置。

4. 当然最重要的还是内在，所以加油吧，虽然是一门很古老的语言，但是存在既有其道理。

5. 以上所说均为 `ISO` 标准，还有一些标准称为 GNU扩展集， `gnu99` 之类的，有兴趣的可以上维基百科自行查询。

0x01-C语言序言

倒是觉得写代码首先不是语法，而是格式，任何时候任何地点，要是自己的代码难以理解，要么你是故意的，要么你就是菜菜

一个难以被人理解的代码在我看来是没有太多的潜力的，但不排除故意为之的情况,也许很多人说这是强迫症，但是无论打开哪一个开源代码，你看到的都将是一个拥有规范的代码文件

也许有人说人不应该被限制，不应该拘泥于小节，但是当一个工程超过一千行，也许不用只需要不到五百行，就能完全暴露出代码规范的重要性，包括缩进，变量命名，接口存放，接口参数的规范之类，听起来似乎很虚：各语言代码规范合集

在我看来C语言的内建语法真是无比简洁，几乎存在既有道理，简洁不代表着不强大，强大的某些地方在近来渐渐复苏的Lisp身上也有体现。

`if, for, while, switch`

组成了每个C程序的半壁江山

`" + " " - " " * " " / " " % " " = "`

组成了各式各样的算法计数

`">>" "<<" "|" "&" "^" "~" "!"`

让C语言有了更高效的算法以及更奇妙的思路

`struct enum union #define return`

而这些则让C语言在这乱世纷争中站稳了脚跟，并且一枝独秀

`"{}" "()"`

让代码不再无序混乱

`"type * " "&" "()" "->"`

让C语言在这个世界无处不在 `" . " "[ ]" " < " " > " " == "`

还记得他们吗？我想这一辈子都忘不了了

0x02-编程带给我的

是快乐而不是痛苦，如果你觉得编程痛苦，请放下你手头的工作，找找自己真正想要的，无论从什么角度来看，你都应该放弃令你痛苦的事情，花上三杯茶的时间，看看自己的心到底喜欢什么。

C语言可谓是让一个程序员最难以感受到自己进步的编程语言，一个黑窗口就让无数程序员再也走不出来。或者迷失，或者停滞不前，或者放弃，一个人最恐惧无助，甚至彷徨的时候，就是在努力之后却感受不到自己在进步，努力的白费是所有人不愿意看到的，但是太多人就着所谓前途而奋不顾身的投入这个事业，他们也许对计算机完全没有喜爱之心，埋头苦干，世人皆称爱读书的好孩子，但是这意义又在何处？即使最后你领着你觉得高的工资，站在了同学，朋友的前方，依然发现自己并没有得到满足，在我看来，让自己开心的才是最好的，不适合你的永远是最差的，即便能带来利益？何不花三杯茶的时间，想想自己到底适合何处。

在C语言的道路上，囊括了许多道天堑，并不是说这门语言比其他语言难，相反它十分符合人类的思维逻辑，但就是因为它存在的时间太久远，普通的使用于它于世界已经无甚大用，在现在这个高级语言遍地走的时代里，有用的只是那些将C语言发挥到极限的工程，不再是小窗口中写一个数据结构，一个算法，也许你觉得徒手写出一棵红黑树很了不起了？那也就是做成一个字典树，在一个浩大的工程中，一个虽重要却不起眼的小部分罢了，学完所有语法，却不知所措接下去该怎么做？有心人在无尽的探索之后发觉，啊！标准库！啊算法！嗯对了，还有各种各样的第三方扩展，以后呢？啊！操作系统！然而自学的路上充满着坎坷，艰辛，无助，烦恼，那又如何？喜欢就好。

所谓师傅领进门，修行在个人，这句话在我看来有两个重要点，却是现在大学生几乎缺失的。师傅一词告诉我们，要不耻下问，要善于询问，而不是伸手即来思想，"提问的智慧"在我看来是一门很重要的课程，特别是在当今信息时代。而更重要的是，先入为主的思想是极其可怕的，在这两年的自学历程里，见过太多后来者居上的事迹，当你一直认为自己一定比后辈强时，你就注定输了，所以不耻下问才是最重要的。但是如果师傅是那么容易找到的，那就不会有学校了，个人指的并不是孤军奋战，而是要善于自己发现问题，努力解决问题，这个过程可能少不了请教他人

编程可以是一种信仰，至少在我认为是这样的，把它当作信仰的人，它就能给你快乐，给你充实，当然也不要忘了现实，虽然现实中总是少不了加班的羁绊，但是如果是眞心喜爱编程，又怎么会被这些困难所打败？但是C语言眞的不是一门容易精通的语言。

0x03-C代码

```
#include <stdio.h>
int main(void)
{
    printf("That is Right Style\n");
    return 0;
}
```

在一个标准的C语言程序中，最特殊的莫过于main函数了，而说到底它就是一个函数而已，仅仅因为它地位特殊拥有第一执行权力，换句话说，难道因为一个人是省长它就不是人类了？所以函数该有的它都应该有，那么函数还有什么呢？

函数大体上分为内联函数(C99)(内联函数并非C++专属，C语言亦有，具体见前方链接)和非内联的普通函数，它们之间有一个很明显的特点(一般情况下)，那就是不写原型直接在main函数上方定义，即使不加'inline'关键字，也能被编译器默认为内联函数，但之后带来的某些并发问题就不是编译器考虑的了。

普通函数正确的形式应该为声明与定义分离，声明就是一个函数原型，函数原型应该有一个函数名字，一个参数列表，一个返回值类型和一个分号。定义就是函数的内在，花括号内的就是函数的定义:

```
//...
int function(int arg_1, float arg_2);
//...
int main(int argc, char* argv[])
{
  int output = function(11, 22.0);
  printf("%d\n",output);
  return 0;
}

int function(int arg_1, float arg_2)
{
 int    return_value  = arg_1;
 float  temp_float    = arg_2;
 return return_value;
}
```

依上所述，当非必要时，在自己编写函数的时候请注意在开头(main函数之前)写上你的函数的原型，并且在末尾(`main` 函数之后)写上你的函数定义，这是一个很好的习惯以及规范。所谓代码整洁之道，就是如此。

函数的另一种分类是，有返回值和无返回值，返回值的类型可以是内建**(build-in)**的也可以是自己定义的(`struct, union` 之类)，无返回值则是 `void` 。

1. 为什么我们十分谴责 `void main()` 这种写法？因为这完全是中国式教育延伸出来的谭式写法，**main**函数的返回值看似无用，实际上是由操作系统接收，在Windows操作系统下也许无甚"大碍"(实际上有),当你使用Linux的过程中你会清晰的发现一个C语言程序的**main**返回值关系到一个系统是否能正常，高效的运行，这里稍微提一句， `0` 在Linux程序管道通信间代表着无错可行的意思。所以请扔掉 `void main` 这种写法。
2. 为什么我们对 `main()` 这种省略返回值的写法置有微词？能发明这种写法的人，必定是了解了，在C语言中，如果一个函数不显式声明自己的返回值，那么会被缺省认为是 `int` ，但这一步是由编译器掌控，然而C语言设计之初便是让我们对一切尽可能的掌握，而一切不确定因子我们都不应该让它存在。其次有一个原则，能自己做的就不要让编译器做。

3. 为什么我们对参数放空置有不满(int main())?在C语言中，一个函数的参数列表有三种合法形态:

```
int function();
int function(void);
int function(int arg_n);
int function(int arg_n, ...);
```

第一种代表拥有未知个参数,第二种代表没有参数，第三种代表有一个参数，第四种代表拥有未知个参数，并且第一个参数类型为int,未知参数在C语言中有一个解决方案就是，可变长的参数列表，具体参考C标准库，在此我们解释的依据就是，我们要将一切都掌控在自己的手中，我们不在括号内填写参数，代表着我们认为一开始的意思是它为空，正因此我们就应该明确说明它为void,而不该让它成为一个未知参数长度的函数，如此在你不小心传入参数的时候，编译器也无法发现错误。

4. `int main(int argc, char* argv[])` 和 `int main(void)` 才是我们该写的C语言标准形式
对于缩进，除了编译器提供的符号缩进之外，我们可以自己给自己一个规范(请少用或者不用Tab)，比如每一块代码相教上一个代码块有4格的缩进。
对于学习C语言，请使用.c文件以及C语言编译器练习以及编写C程序，请不要再使用C++的文件编写C语言程序，并且自圆其说为了效率而使用C++的特性在C语言中，我们是祖国的下一代，是祖国的未来，请不要让自己毁在当下，珍爱编程，远离清华大学出版社

之所以如此叙述，并不是因为情绪，而是当眞如此，下方代码:

```
/*file: test.c*/
#include <stdio.h>
#define SIZES 5
int main(void)
{
    int* c_pointer = malloc(SIZES * sizeof(int));
    /*发生了一些事情*/
    free(c_pointer);
    return 0;
}
```

这是一段标准的C语言程序，但是它能在C++个编译器下编译运行吗？换句话说当你将文件扩展名由 `.c` 改为 `.cpp` 之后，它能编译通过吗？答案是不能。

为什么？答案是C++并不支持 `void*` 隐式转换为其他类型的指针，但是C语言允许。还有许许多多C于C++不相同的地方，兴许有人说C++是C的超集，但我并不这么认为，一门语言的出现便有它的意义所在，关键在于我们如何发挥它的最大优势，而不是通过混淆概念来增强实用性

5. 程序式子的写法

一个人活在世界上，时时刻刻都注意着自己的言行举止，而写程序也是如此，对于一个规范的能让别人读懂的程序而言，我们应该尽可能减少阻碍因子，例如:

```
int main(void)
{int complex_int=100;
int i,j,k,x;
for(int temp=0;temp<complex_int;++temp){k=temp;
x=k+complex_int;}
printf(complex_int="%d is k=%d x=%d\n",complex_int,k,x);
return 0;}
```

对于上述的代码，我总是在班级里的同学手下出现，但这段代码除了让别人困惑以外，自己在调试的时候也是十分不方便，每每遇到问题了，即便IDE提示了在某处错误，你也找不到问题所在，经常有人来问我哪里错了，大部分情况都是少了分号，括号，或者作用域超过，原因在哪？

要是一开始将代码写清楚了，这种情况简直是凤毛麟角，想遇上都难。对于一个代码而言，我们应该注意让其变得清晰。

- 等号两边使用空格：

  ```
  int complex_int = 100;
  ```

- 使用多个变量的声明定义，或者函数声明定义，函数使用时，注意用空格分开变量：

  ```
  int i, j, k, x;//但是十分不建议这么声明难以理解意义的变量
  printf("complex_int = %d is k = %d x = %d\n", complex_int, k, x);
  void present(int arg_1, double arg_2);
  ```

- 对于一个清晰的程序而言，我们要让每一个步骤清晰且有意义，这就要求我们在编写程序的时候尽量能让代码看起来结构化，或者整体化。尽量让每个程序式子为一行，如果有特别的需要让多个式子写在同一行，可以使用 `,` 操作符进行组合，但是会让程序更难理解，日后调试的时候也更难发现错误。

```
/*Style 1*/
for(int temp = 0;temp < complex_int;++temp)
{
    k = temp;
    x = k + complex_int;
}
/*Style 2*/
for(int temp = 0;temp < complex_int;++temp){
    k = temp;
    x = k + complex_int;
}
```

  对于上方的代码，是C语言代码花括号的两种风格，最好能选择其中一种作为自己的编程风格，这样能让你的程序看起来更加清晰，混合使用的利弊并不好说，关键还是看个人风格。

- 对于作用域而言，在C语言中有一个经常被使用的特例，当一个条件语句，或者循环只有一条语句的时候，我们常常省略了花括号 `{}` ，而是仅仅使用一个分号作为结尾，这在很多情况下让代码不再啰嗦：

```
if(pointo_int == NULL)
    fprintf(stderr, "The pointer is NULL!\n");
else
{
    printf("%d\n",*pointo_int);
    pointo_int = pointo_int->next;
}
```

  在这段代码中 `if` 语句下方的代码并没有使用 `{}` 运算符进行指明，但是根据语法，该语句的确是属于 `if` 语句的作用范围内，如果我们此时写上了 `{}` 反而会令代码看起来过于啰嗦。但是有的时候，这条特性并不是那

么的有趣，当使用嵌套功能的时候，还是建议使用 `{}` 进行显式的范围规定，而不是使用默认的作用域：

```
for(int i = 0;i< 10;++i)
    for(int k = 0;k < 10;++k)
        while(flag != 1)
            set_value(arr[i][k]);
```

这段代码，看起来十分简洁，但是确实是一个很大的隐患，当我们要调试这段代码的时候，总是需要修改它的构造，而这就带来了潜在的隐患。所以建议在使用嵌套的时候，无论什么情况，都能使用 `{}` 进行包装。

综上所述，在开始编写一个标准C语言程序的时候，请先把下面这些东西写上:

```
#include <stdio.h>

int main(void)
{
    return 0;
}
```

对于 `main` 的参数，有兴趣的可以查阅我的文章，或者自行谷歌，在此问题上百度也是可以的。

0x04 C代码规范

1. 命名
    - 只要提到代码规范，就不得不说的一个问题。
    - 在一些小的演示程序中，也许费尽心思去构思一个 命名 是一件十分傻的行为，但是只要程序上升到你需要严正设计，思考，复查的层次，你就需要好好考虑 命名 这个问题。
    - 函数命名：
        - C语言中，我们可以让下划线或者词汇帮助我们表达函数功能：
            1. 前缀：
                1. `set` 可以表示设置一个参数为某值
                2. `get` 可以表示获取某一个参数的值
                3. `is` 可以表示询问是否是这种情况
            2. 后缀：
                1. `max/min` 可以表示某种操作的最大(小)次数
                2. `cnt` 可以表示当前的操作次数
                3. `key` 某种关键值

                ```
                size_t get_counts();
                size_t retry_max();
                int    is_empty();
                ```
        - 需要注意的只是，不要让命名过于赘述其义，只简单保留动作以及目的即可，详细功能可以通过文档来进行进一步的解释。
    - 结构体命名：
        - 由于结构体的 标签，不会污染命名，即标签不在命名搜索范围之内，所以可以放心使用：
            1. 有人习惯使用 `typedef` ,而有人喜欢使用 `struct tag obj` ，后者比较多，但是前者也不失为一种好方法，仁者见仁智者见智。

```
/*方法1*/
struct inetaddr_4{
        int    port;
        char * name;
};
struct inetaddr_4 *addr_info;
/*方法2*/
typedef struct _addr{
        int    port;
        char * name;
}inetaddr_4;
inetaddr_4 *addr_info_2;
```

两者同处一个文件内亦不会发生编译错误。

- 变量命名
    - 所有字符都使用小写
    - 含义多的可以用 `_` 进行辅助
    - 以 `=` 为标准进行对齐
    - 类型， 变量名左对齐。
    - 等号左右两端，最少有一个空格。

```
int main(void)
{
    int          counts = 0;
    inetaddr_4   *addr    = NULL;

    return 0;
}
```

为了防止指针声明定义时候出错，将 `*` 紧贴着变量名总不会出错。

```
inetaddr_4   *addr, object, *addr_2;
```

其中 `addr` 和 `addr_2` 是指针，而 `object` 则是一个栈上的完整对象，并不是指针。

        - 全局变量能少用就少用，必须要用的情况下，可以考虑添加前缀 `g_`

```
int g_counts;
```

    - `#define` 命名
        - 所有字符都是用大写，并用 `_` 进行分割
        - 如果多于一个语句，使用 `do{...}while(0)` 进行包裹，防止 `;` 错误。

```
#define SWAP(x, y)          \
do{                         \
    x = x + y;        \
    y = x - y;            \
    x = x - y;            \
}while(0)
```

当然这个交换宏实际上有一点缺陷，在大后方会提出。此处是代码规范，就不重复强调。

    - `enum` 命名
        - 所有字符都是用大写，并用 `_` 进行分割
        - 与 `define` 相比， `enum` 适用于同一类型的常量声明，而不是单一独立的常量。往往出现都是成组。

1. 格式化代码
    - 花括号 `{}`
        - 混合使用符合节俭思想，但会稍微有一点结构紊乱。
        - 单一使用能更好让代码结构清晰。
        - 所谓混合，单一指的是是否一直使用 `{}` 进行代码包裹。
        - 有人认为 当单一语句的时候不必要添加 `{}` ，有的人则习惯添加
        - 当作用域超过一个屏幕的时候，可以适当的使用注释来指明 `{}` 作用域

```
while(1){
    if(tmp == NULL){
        break;
    }
    else if(fanny == 1){
        ... 大概超过了一个屏幕的代码
    } /*else if fanny*/
}/*end while*/
```

如果是代码量少的情况下，但嵌套比较多，也可以使用这个方式进行注释。

- 括号 `()`
  - 有人建议除了函数调用以外，在条件语句等类似情况下使用 `()` 要在关键字后空一格，再接上 `()` 语句，对于这一点，我个人习惯是不空格，但总有这种说法。

```
if (space == NULL) {
    /**TODO**/
}
while(1){
    /**我习惯于如此写**/
}
strcpy(str1, str2); /**第一种写法是为了和函数调用写法进行区分**/
return 0;
```

- `switch`
  - 一定要放一个 `default` 在最后，即使它永远不会用到。
  - 每个 `case` 如果需要使用新变量，可以用 `{}` 包裹起来，并在里面完成所有操作。

```
switch(...)
{
    case 1:
        /**TODO**/
        break;

    case 2:
    {
        int new_vari;
        /**创建新变量则用 {} 包裹起来**/
    }
    break;

    default:
        call_error();
}
```

- `goto`
  - 虽然许多人，许多书都提醒不再使用 `goto` 关键字，而是使用 `setjmp` 和 `longjmp` 来取代它，但是这还是那句话，仁者见仁智者见智，如果 `goto` 能够让代码清晰，那何乐而不为呢，这个观点也是最近才体会到的（并非我一己之言）。
  - 具体使用可以查询官方文档。
- 语句
  - 应该让完整的语句在每一行中，只出现一次。
  - 对于变量声明定义亦是如此
  - 原因是这样能让文档更有针对性
- 头文件保护
  - 对于头文件而言，在一个程序中有可能被多次包含(`#include`)，如果缺少头文件保护，则会发生编译错误
  - 不要将 `_` 作为宏的开头或者结尾。

```
#ifndef VECTOR_H_INCLUDE
    #define VECTOR_H_INCLUDE
        /**TODO**/
    #endif
```

2. 宏
    - C语言的宏有诸多弊端，所以尽量使用 `inline` 函数来代替宏。在大后方会有解释
    - 但是，请不要因此抛弃了宏，比如在 `C11` 中有一个新兴的宏。
3. 变量
    - 第一时刻初始化所有所声明的变量，因为这么做总没有坏处，而且能减少出错的可能。
4. 函数
    - 函数应该尽可能的短小，一个ANSI屏幕的为最佳。
5. 如果某个循环带着空语句，使用 `{}` 进行挂载，以免出现意外。

   ```
   while(*is_end++ != '\0')
   {
       ;
   }
   ```

   虽然是空的循环体，但是写出来以免造成误循环。

6. 尽量不要让函数返回值直接作为条件语句的判断，这样会极大降低可读性

   ```
   if(is_eof(file) == 0)
       好过
   if(!is_eof(file))
   ```

7. 不要为了方便或者一点点的所谓速度提升(也许根本没有)，而放弃可读性，使用嵌入式的赋值语句

```
int add = 10;
int num = 11;
int thr = 20;
add = add + thr;
num = add + 20;
    不要写成
num = (add = add + thr) + 20;
```

浮点数

- 万万记住不要再使用浮点数比较彼此是否相等或不等。
- 如果把浮点数用在离散性的数据上，比如循环计数器，那就等死吧。

其他

- 使用 `#if` 而不是 `#ifdef`
- 可以使用 `define()` 来代替 `#ifdef` 的功能

```
#if !define(USERS_DEFINE)
    #define USERS_DEFINE ...
#endif
```

- 对于某些大段需要消除的代码，我们不能使用注释 `/**/` ，因为注释不能内嵌着注释(`//` 除外)，我们可以使用黑魔法：

```
#if NOT_DECLARATION
    /**想要注释的代码**/
#endif
```

- 不要使用纯数字
    - 意味着，不在使用毫无标记的数字，因为可能你过了几个月再看源代码的时候，你根本不知道这个数字代表着什么
    - 而应该使用 `#define` 给它一个名字，来说明这个数字的意义。

0x05-C语言变量

C语言在明面上将数的变量分为两类，整型变量以及浮点数，对应着现实世界的整数和小数。

- 首先是整数，使用了这么多的C语言之后，每当在使用整数之时都会将其想象成二进制的存在，而不是十进制。原因在于，这是程序的本质所在，稍有研究编译器工作原理的都会发现，在编译器处理乘法乃至除法的时候，优秀的编译器总会想方设法的加快程序的速度，毫无疑问在所有运算中移位运算是最快速的"乘法"以及"除法":

  `1<<2 == 4` , `8>>2 == 2`

  而正常一个乘法相当于十数次的加法运算的时间消耗，移位则不用(除法的消耗更大，但是随着**CPU**的进步，这些差距正在逐渐缩小，就目前来看依旧是有着不小的差距但无论如何优化，乘法时间都会大于加法)。正如前面所说，C语言设计之初便是给了程序员所有的权利，而程序员要做的就是掌控所有能掌控的，即便是数的计算亦是如此，比如在优秀的编译器看来:

  `2*7 ====> (2<<3) - 2`

  `5*31 ====> (5<<5) - 5`

  毫无疑问经过编译器优化后的代码此前者要快许多。这就是为什么我们要将一个数看作二进制，这不仅仅是表面，而是要在深层次的认为它是二进制，总体来说C语言的整型是非常简洁明了的总体分为 有符号 和 无符号，很好理解只需要注意不要让无符号数进行负数的运算，这里有一个原则，可以很好的规避这种无意之过，不把无符号类型变量和有符号类型变量放于同一运算中，时刻记得保持式子的类型一致是设计时的保障。

- 浮点数，由于实数域可以看作稠密的，故除了整数以外，还有无数的小数，而小数在计算机中如何表示？一种无限的状态是无法在计算机中被精确表示，所以有了浮点法，关于浮点法可以参考书籍《深入理解计算机系统》。

  这里介绍的是在C语言中我们应该如何正确使用浮点数？很多人(包括我)在初作之时总是想当然的以为计算机是无所不能的，连人类都无法完全表达出来的小数计算机一定可以，实际上并非如此，在这里我可以说，计算机只是近似表达，而最大的忌讳的便是将两个浮点数进行比较，此处介绍一种浮点数常用的比较方法，精确度法:

```
#define DISTANCE 0.00000001
...
float f_x_1 = 20.5;
float f_x_2 = 19.5;
if(f_x_1 - f_x_2 < DISTANCE)
    printf("They are Equal\n");
else
    printf("Different\n");
```

所以说，在很大程度上，当你在程序中使用了浮点数，又直接使用浮点数进行比较，却发现始终无法达到预期效果，那么你可以检查一下，是否是这个原因，在这一点上，不得不说是C语言的一个缺憾。

- 指针变量，是一种比较特别的变量，以至于总是对它进行特别对待。这里有几个原则:
    - 两个不相关的指针进行加减操作是无意义的
    - 始终确保自己能够找到分配的内存
    - 无论何时何地何种情况，都要记住，不使用未初始化的指针，不让未使用的内存持续存在。

  指针在不同位的操作系统上的大小是不一样的，但是在同一个操作系统下，无论什么类型的指针都是相同大小，这涉及到指针的寻址问题，**(**题外话**:C**语言的寻址实际上使用了汇编语言的间接寻址，有兴趣的可以自行尝试，方法之一，使用**gcc**编译器的汇编选项，产生汇编代码，进行一一比对**)**，对于寻址一个笼统一些的说法便是
  `4Byte = 32bit`
  `2^32 = 4G`
  所以32位的操作系统下C语言指针:

```
...
size_t what = sizeof(void*);
printf("%d", what);
...
```

  输出: `$root@mine: 4`
  对于大部分使用者来说，指针主要用来降低内存消耗以及提高运算效率的，这里设计许多学问，我也无法一一展示，比较有意思也常用的两个东西便是递增

以及语法糖: `++, ->`

```
...
int dupli_of_me[10] = {0};//也可以使用库函数memset()进行置0
int *point_to_me = dupli_of_me;
int me = 100;
while(point_to_me < (dupli_of_me + 10))
    *point_to_me++ = me;
```

其中 `*point_to_me++ = me;` 在C语言应用广泛它相当于是:

```
*point_to_me = me;
point_to_me++;
```

的语法糖，对于 `++` ，在非必要的情况下，请使用前缀递增，而非后缀递增，原因是消耗问题，仔细想想这两种递增的区别在何处？
前缀递增总是在原数上进行递增操作，然而后缀递增呢？它首先拷贝一份原数放于别处，并且递增这份拷贝，在原数进行的操作完毕后，将这份拷贝再拷贝进原数取代它，此中的操作涉及的更多，所以在非必要的情况下，请使用前缀递增而不是后缀递增(递减也是同样的道理)
`->` 则是在结构体上使用的非常广泛:

```
typedef struct data{
    int test;
    struct data* next;
}my_struct;
...
my_struct temp;
my_struct *ptemp = &temp;
ptemp->test = 100;
ptemp->next = NULL;
if(temp.test == 100)
    printf("Correctly!\n");
else
    printf("That is impossible!\n");
...
```

可以很清楚的看出其实 `ptemp->test` 便是 `(*ptemp).test` 的语法糖

- 变量限定

`const` 是最常用的变量限定符，它的意思是告诉编译器，这个变量或者对象在初始化以后不能被改变，常用它来保护一些必要的返回值，参数以及常量的定义。

`volatile` 这个关键字常常被C语言教材所忽略，它很神秘。实际上确实如此，他的作用的确很神秘：一旦使用了，就是告诉编译器，即使这个变量没有被使用或修改其他内存单元，它的值也可能发生变化。通俗的说就是，告诉编译器，不要把你的那一套优化策略用在我身上。

```
    /* 此时我们将编译器优化等级提高到 -O2 */
    int          test_num   = 100; //测试一个迭代加法
    int          nor_result = 0;
    volatile int vol_result = 0;
    /* 测试无volatile限定下，该程序的耗时 */
    for(int i = 0;i < 10000;++i)
        for(int j = 0;j < 10000;++j)
                nor_result += test_num;
```

接下来就是测试 `volatile` 限定下的代码

```
    for(int i = 0;i < 10000;++i)
        for(int j = 0;j < 10000;++j)
                vol_result += test_num;
```

在使用一些手段后，得到运行时间，可以很清晰的看出差别，在我的机器上， `i5-4CPU` ，得到的结果是后者比前者慢大概十五倍。 从某一些方向上证明了，volatile的一些作用，比如调试的时候，或者一些特殊用途。涉足不多，故不记录。

- 变量说明

`extern` 用于将不同文件的，带有外部链接性的变量引用到本文件中。所谓外部链接性就是可以被除本文件外的其他文件"看见"的变量，如全局变量，使用方法：

```
/* 以下为一个工程内可见 */
/*file1.c*/
int glo_show;//对于该全局变量来说，它们在声明时无初始化，则默认初始
为0
int glo_print = 10;//声明定义完成后，自动分配内存以存储信息
...

/* file2.c */
extern glo_print; //仅仅是引用名字，并不会额外分配空间
                  //所以，只需要写正确变量名字即可，后方的初始化无
须完全
                  //因为变量的初始化定义只能有一次。

void print()
{
    printf("The Globle Value is %d \n", glo_print);
}
```

`auto` 可以姑且忽略，因为没有什么实际意义。

- 变量获取

格式化输入输出在C语言的初学中使用的比较频繁，但是到后期会发现，由于**I/O**操作过于消耗资源，换句话来说就是会极大影响程序的执行效率，会渐渐的在发行版程序中消除。

  - 常见格式化输入标准函数： `sacnf` , `fscanf` , `sscanf`

    对于常见的使用不赘述，有两种比较不常见的格式： `%[]` 和 `%*` ，前者是用于限制读取类型，常见于字符串的过滤(不是真正的过滤)

    ```
    scanf("%d %[a-z]", &tmp, str);
    scanf("%d %[^i]", &tmp, str);
    scanf("%d %[^,]", &tmp, str);
    ```

    假设输入的是： `22 hello,string to me!`
    读取到的分别为: `22 hello` 和 `22 hello,str` 和 `22 hello`
    后者则是忽略第一个输入：

```
    scanf("%*d %d", &tmp);
```

假设输入的是：`22 33`
读取到的则是：`33`
其中开头的 `%*d` 忽略的输入，必须和其类型匹配，例如输入：`string 33` 则会读取失败。
也可以将其解读为文件宽度，例如在使用 `printf` 格式化输出的时候:

```
char str[10] = "dir";
printf("%*s%s",4 ,"" , str);
/* 输出:    dir */ 四个空白占位
```

但是实际上 `scanf` 并不太好用，所谓的好用指的是功能上以及设计上的缺陷，总是让很多人摸不着头脑的出了错，往往很难调试。例如它会将每一行输入的 `\n` 保留在输入流里面，这个缺陷导致如果不明所以得人将其与其他的输入函数，例如 `fgets` 或者 `gets` 配合会出现差错。

第二部分

> C语言核心知识 下

0x05-C语言指针:(Volume-1)

这似乎是一个很凝重的话题，但是它真的很有趣。

1. 指针是指向某一类型的东西，任何一个整体，只要能称为整体就能拥有它自己的独一无二的指针类型，所以指针的类型其实是近似无穷无尽的

2. 函数名在表达式中总是以函数指针的身份呈现，除了取地址运算符以及 **`sizeof`**

3. C语言最晦涩难明的就是它复杂的声明**:** **`void (*signal(int sig, void (*func)(int)))(int)`** **,**试试着把它改写成容易理解的形式

4. 对于指针，尽最大的限度使用 **`const`** 保护它，无论是传递给函数，还是自己使用

先来看看一个特殊的指针，姑且称它为指针，因为它依赖于环境: `NULL` ，是一个神奇的东西。先附上定义，在编译器中会有两种NULL(每种环境都有唯一确定的NULL):

```
#define NULL 0
#define NULL ((void*)0)
```

有什么区别吗？看起来没什么区别都是 `0` ，只不过一个是常量，一个是地址为0的指针。

当它们都作为指针的值时并不会报错或者警告，即编译器或者说C标准认为这是合法的:

```
int* temp_int_1 = 0; //无警告
int* temp_int_2 = (void*)0; //无警告
int* temp_int_3 = 10; //出现警告
```

为什么？为什么 `0` 可以赋值给指针，但是 `10` 却不行？他们都是常量。

因为C语言规定当处理上下文的编译器发现常量 `0` 出现在指针赋值的语句中，它就作为指针使用，似乎很扯淡，可是却是如此。

回到最开始，对于 `NULL` 的两种情况，会有什么区别？拿字符串来说，实际上我是将字符数组看作是C风格字符串。

在C语言中，字符数组是用来存储一连串有意义的字符，默认在这些字符的结尾添加 `'\0'` ，好这里又出现了一个0值。

对于某些人，在使用字符数组的时候总是分不清楚 `NULL` 与 `'\0'` 的区别而误用，在字符数组的末尾使用 `NULL` 是绝对错误的！虽然它们的本质都是常量0，但由于位置不同所以含义也不同。

开胃菜已过

对于一个函数，我们进行参数传递，参数有两种形式: 形参与实参

```
int function(int value)
{
        /*...*/
}
//...
function(11);
```

其中， `value` 是形参， `11` 是实参，我们知道场面上，C语言拥有两种传递方式: 按值传递和按址传递，但是你是否有认真研究过？这里给出一个实质，其实C语言只有按值传递，所谓按址传递只不过是按值传递的一种假象。至于原因稍微一想便能明白。

对于形参和实参而言两个关系紧密，可以这么理解总是实参将自己的一份拷贝传递给形参，这样形参便能安全的使用实参的值，但也带给我们一些麻烦，最经典的交换两数

```
void swap_v1(int* val_1, int* val_2)
{
    int temp = *val_1;
    *val_1 = *val_2;
    *val_2 = *val_1;
}
```

这就是所谓的按址传递，实际上只是将外部指针(实参)的值做一个拷贝，传递给形参 `val_1` 与 `val_2` ，实际上我们使用:

```
#define SWAP_V2(a, b) (a += b, b = a - b, a -= b)
#define SWAP_V3(x, y) {x ^= y; y ^= x; x ^= y}
```

试一试是不是很神奇，而且省去了函数调用的时间，空间开销。上述两种写法的原理实质是一样的。

但是，动动脑筋想一想，这种写法真的没有瑕疵吗？如果输入的两个参数本就指向同一块内存，会发生什么？

```
...
int test_1 = 10, test_2 = 100;
SWAP_V2(test_1, test_2);
printf("Now the test_1 is %d, test_2 is %d\n", test_1, test_2);
.../*恢复原值*/
SWAP_V2(test_1, test_1);
printf("Now the test_1 is %d\n", test_1);
```

会输出什么？:

```
$: Now the test_1 is 100, test_2 is 10
$: Now the test_1 is 0
```

对，输出了0，为什么？稍微动动脑筋就能相通，那么对于后面的 `SWAP_V3` 亦是如此，所以在斟酌之下，解决方案应该尽可能短小精悍:

```
static inline void swap_final(int* val_1, int* val_2)
{
    if(val_1 == val_2)
        return;
    *val_1 ^= *val_2;
    *val_2 ^= *val_1;
    *val_1 ^= *val_2;
}
#define SWAP(x, y) \
do{                    \
    if(&x == &y)    \
        break;         \
    x ^= y;        \
    y ^= x;        \
    x ^= y;        \
}while(0)
```

这便是目前能找到最好的交换函数，我们在此基础上可以考虑的更深远一些，如何让这个交换函数更加通用？即适用范围更大？暂不考虑浮点类型。 提示：可用 `void*`

与上面的情况类似，偶尔的不经意就会造成严重的后果:

```
int combine_1(int* dest, int* add)
{
    *dest += *add;
    *dest += *add;
    return *dest;
}
int combine_2(int* dest, int* add)
{
    *dest += 2 * (*add);//在不确定优先级时用括号是一个明智的选择
    return *dest;
}
```

上述两个函数的功能一样吗？恩看起来是一样的

```
int test_3 = 10, test_4 = 100;

combine_1(&test_3, &test_4);
printf("After combine_1, test_3 = %d\n",test_3);
.../*恢复原值*/
combine_2(&test_3, &test_4);
printf("After combine_2, test_3 = %d\n",test_3);
```

输出

`$: After combine_1, test_3 = 210`

`$: After combine_2, test_3 = 210`

如果传入两个同一对象呢？

```
... /*恢复test_3原值*/
combine_1(&test_3, &test_3);
printf("After second times combine_1, test_3 = %d\n",test_3);
...
combine_2(&test_3, &test_3);
printf("After second times combine_2, test_3 = %d\n",test_3);
```

输出

`$: After second times combine_1, test_3 = 40`

`$: After second times combine_2, test_3 = 30`

知道真相总是令人吃惊，指针也是那么令人又爱又恨。

- **C99** 标准之后出现了一个新的关键字， `restrict` ，被用于修饰指针，它并没有太多的显式作用，甚至加与不加，在 你自己 看来，效果毫无区别。但是反观标准库的代码中，许多地方都使用了该关键字，这是为何
  - 首先这个关键字是写给编译器看的
  - 其次这个关键字的作用在于辅助编译器更好的优化该程序(后方文章会有介绍)
  - 最后，如果不熟悉，绝对不要乱用这个关键字。

关于数组的那些事

数组和指针一样吗？

不一样

要时刻记住，数组与指针是不同的东西。但是为什么下面代码是正确的？

```
int arr[10] = {10, 9, 8, 7};
int* parr  = arr;
```

我们还是那句话，结合上下文，编译器推出 `arr` 处于赋值操作符的右侧，默默的将他转换为对应类型的指针，而我们在使用 `arr` 时也总是将其当成是指向该数组内存块首位的指针。

```
//int function2(const int test_arr[10]
//int function2(const int test_arr[]) 考虑这三种写法是否一样
int function2(const int* test_arr)
{
    return sizeof(test_arr);
}
...
int size_out = sizeof(arr);
int size_in  = function2(arr);

printf("size_out = %d, size_in = %d\n", size_out, size_in);
```

`输出: size_out = 40, size_in = 8`

这就是为什么数组与指针不同的原因所在，在外部即定义数组的代码块中，编译器通过上下文发觉此处arr是一个数组，而 `arr` 代表的是一个指向**10**个**int**类型的数组的指针，只所谓最开始的代码是正确的，只是因为这种用法比较多，就成了标准的一部分。就像世上本没有路，走的多了就成了路。"正确"的该怎么写

```
int (*p)[10] = &arr;
```

此时 `p` 的类型就是一个指向含有**10**个元素的数组的指针,此时 `(*p)[0]` 产生的效果是 `arr[0]` ，也就是 `parr[0]` ，但是 `(*p)` 呢？这里不记录，结果是会溢出，为什么？

这就是数组与指针的区别与联系，但是既然我们可以使用像 `parr` 这样的指针，又为什么要写成 `int (*p)[10]` 这样丑陋不堪的模式呢？原因如下:

- 回到最开始说过的传递方式，按值传递在传递 `arr` 时只是纯粹的将其值进行传递，而丢失了上下文的它只是一个普通指针，只不过我们程序员知道它指向了一块有意义的内存的起始位置，我想要将数组的信息一起传递，除了额外增加一个参数用来记录数组的长度以外，也可以使用这个方法，传递一个指向数组的指针 这样我们就能只传递一个参数而保留所有信息。但这么做的也有限制:对于不同大小，或者不同存储类型的数组而言，它们的类型也有所不同

  ```
  int arr_2[5];
  int (*p_2)[5] = &arr_2;
  float arr_3[5];
  float (*p_3)[5] = &arr_3;
  ```

  如上所示，指向数组的指针必须明确指定数组的大小，数组存储类型，这就让指向数组的指针有了比较大的限制。

- 这种用法在多维数组中使用的比较多，但总体来说平常用的并不多，就我而言，更倾向于使用一维数组来表示多维数组，实际上诚如前面所述，C语言是一个非常简洁的语言，它没有太多的废话，就本质而言C语言并没有多维数组，因为内存是一种线性存在，即便是多维数组也是实现成一维数组的形式。

  - 就多维数组在这里解释一下。所谓多维数组就是将若干个降一维的数组组合在一起，降一维的数组又由若干个更降一维的数组组合在一起，直到最低的一维数组，举个例子:

    int dou_arr[5][3]; 就这个二维数组而言，将5个每个为3个 `int` 类型的数组组合在一起，要想指向这个数组该怎么做？

    ```
    int (*p)[3]        = &dou_arr[0];
    int (*dou_p)[5][3] = &dou_arr;
    int (*what_p)[3]   = dou_arr;
    ```

    实际上多维数组只是将多个降一维的数组组合在一起，令索引时比较直观而已。当真正理解了内存的使用，反而会觉得多维数组带给自己更多限制对于第三句的解释，当数组名出现在赋值号右侧时，它将是一个指针，类

> 型则是 指向该数组元素的类型，而对于一个多维数组来说，其元素类型则是其降一维数组，即指向该降一维数组的指针类型。这个解释有点绕，自己动手写一写就好很多。

对于某种形式下的操作，我们总是自然的将相似的行为结合在一起考虑。考虑如下代码:

```
int* arr_3[5] = {1, 2, 3, 4, 5};
int* p_4      = arr_3;

printf("%d == %d == %d ?\n", arr_3[2], *(p_4 + 2), *(arr_3 + 2))
;
```

`输出: 3 == 3 == 3 ?` 实际上对于数组与指针而言， `[]` 操作在大多数情况下都能有相同的结果，对于指针而言 `*(p_4 + 2)` 相当于 `p_4[2]` ，也就是说 `[]` 便是指针运算的语法糖，有意思的是 `2[p_4]` 也相当于 `p_4[2]` ， `"Iamastring"[2] == 'm'` ，但这只是娱乐而已，实际中请不要这么做，除非是代码混乱大赛或者某些特殊用途。 在此处，应该声明的是这几种写法的执行效率完全一致，并不存在一个指针运算便快于 `[]` 运算，这些说法都是上个世纪的说法了，随着时代的发展，我们应该更加注重代码整洁之道

在此处还有一种奇异又实用的技巧，在char数组中使用指针运算进行操作，提取不同类型的数据，或者是在不同类型数组中，使用 `char*` 指针抽取其中内容，才是显示指针运算的用途。但在使用不同类型指针操作内存块的时候需要注意，不要操作无意义的区域或者越界操作。

实际上，最简单的安全研究之一，便是利用溢出进行攻击。

Advance:对于一个函数中的某个数组的增长方向，总是向着返回地址的，中间可能隔着许多其他自动变量，我们只需要一直进行溢出试验，直到某一次，该函数无法正常返回了！那就证明我们找到了该函数的返回地址存储地区，这时候我们可以进行一些操作，例如将我们想要的返回地址覆盖掉原先的返回地址，这就是所谓的溢出攻击中的一种。

0x05-C语言指针(Volume-2)

内存的使用的那些事儿

你一直以为你操作的是真实物理内存，实际上并不是，你操作的只是操作系统为你分配的资格虚拟地址，但这并不意味着我们可以无限使用内存，那内存卖那么贵干嘛，实际上存储数据的还是物理内存，只不过在操作系统这个中介的介入情况下，不同程序窗口(可以是相同程序)可以共享使用同一块内存区域，一旦某个傻大个程序的使用让物理内存不足了，我们就会把某些没用到的数据写到你的硬盘上去，之后再使用时，从硬盘读回。这个特性会导致什么呢？假设你在Windows上使用了多窗口，打开了两个相同的程序:

```
...
int  stay_here;
char tran_to_int[100];
printf("Address: %p\n", &stay_here);

fgets(tran_to_int, sizeof(tran_to_int), stdin);
sscanf(tran_to_int, "%d", &stay_here);

for(;;)
{
    printf("%d\n", stay_here);
    getchar();
    ++stay_here;
}
...
```

对此程序(引用前桥和弥的例子)，每敲击一次回车，值加1。当你同时打开两个该程序时，你会发现，两个程序的 `stay_here` 都是在同一个地址，但对它进行分别操作时，产生的结果是独立的！这在某一方面验证了虚拟地址的合理性。虚拟地址的意义就在于，即使一个程序出现了错误，导致所在内存完蛋了，也不会影响到其他进程。对于程序中部的两个读取语句，是一种理解C语言输入流本质的好例子，建议查询用法，这里稍微解释一下:

- 通俗地说，fgets将输入流中由调用起，`stdin` 输入的东西存入起始地址为 `tran_to_int` 的地方，并且最多读取 `sizeof(tran_to_int)` 个，并在后方 `sscanf` 函数中将刚才读入的数据按照 `%d` 的格式存入 `stay_here` ，这就是C语言一直在强调的流概念的意义所在，这两个语句组合看起来也就是读取一个数据这么简单，但是我们要知道一个问题，一个关于 `scanf` 的问题

  ```
  scanf("%d", &stay_here);
  ```

  这个语句将会读取键盘输入，直到回车之前的所有数据，什么意思？就是回车会留在输入流中，被下一个输入读取或者丢弃。这就有可能会影响我们的程序，产生意料之外的结果。而使用上当两句组合则不会。

函数与函数指针的那些事

事实上，函数名出现在赋值符号右边就代表着函数的地址

```
int function(int argc){ /*...*/
}
...
int (*p_fun)(int) = function;
int (*p_fuc)(int) = &function;//和上一句意义一致
```

上述代码即声明并初始化了函数指针， `p_fun` 的类型是指向一个返回值是**int**类型，参数是**int**类型的函数的指针

```
p_fun(11);
(*p_fun)(11);
function(11);
```

上述三个代码的意义也相同，同样我们也能使用函数指针数组这个概念

```
int (*p_func_arr[])(int) = {func1, func2,};
```

其中 `func1,func2` 都是返回值为 `int` 参数为 `int` 的函数，接着我们能像数组索引一样使用这个函数了。

Tips: 我们总是忽略函数声明，这并不是什么好事。

- 在C语言中，因为编译器并不会对有没有函数声明过分深究，甚至还会放纵，当然这并不包含内联函数(**inline**)，因为它本身就只在本文件可用。
- 比如，当我们在某个地方调用了一个函数，但是并没有声明它：

  ```
  CallWithoutDeclare(100); //参数100为 int 型
  ```

  那么，C编译器就会推测，这个使用了 `int` 型参数的函数，一定是有一个 `int` 型的参数列表，一旦函数定义中的参数列表与之不符合，将会导致参数信息传递错误(编译器永远坚信自己是对的！)，我们知道C语言是强类型语言，一旦类型不正确，会导致许多意想不到的结果(往往是Bug)发生。

- 对函数指针的调用同样如此

C语言中malloc的那些事儿

我们常常见到这种写法:

```
int* pointer = (int*)malloc(sizeof(int));
```

这有什么奇怪的吗？看下面这个例子:

```
int* pointer_2 = malloc(sizeof(int));
```

哪个写法是正确的？两个都正确，这是为什么呢，这又要追求到远古C语言时期，在那个时候， `void*` 这个类型还没有出现的时候， `malloc` 返回的是 `char*` 的类型，于是那时的程序员在调用这个函数时总要加上强制类型转换，才能正确使用这个函数，但是在标准C出现之后，这个问题不再拥有，由于任何类型的指针都能与 `void*` 互相转换，并且C标准中并不赞同在不必要的地方使用强制类型转换，故而C语言中比较正统的写法是第二种。

题外话: C++中的指针转换需要使用强制类型转换，而不能像第二种例子，但是C++中有一种更好的内存分配方法，所以这个问题也不再是问题。

Tips:

- C语言的三个函数 `malloc` , `calloc` , `realloc` 都是拥有很大风险的函数，在使用的时候务必记得对他们的结果进行校验，最好的办法还是对他们进行再包装，可以选择宏包装，也可以选择函数包装。

- `realloc` 函数是最为人诟病的一个函数，因为它的职能过于宽广，既能分配空间，也能释放空间，虽然看起来是一个好函数，但是有可能在不经意间会帮我们做一些意料之外的事情，例如多次释放空间。正确的做法就是，应该使用再包装阉割它的功能，使他只能进行扩展或者缩小堆内存块大小。

指针与结构体

```
typedef struct tag{
        int  value;
        long vari_store[1];
}vari_struct;
```

乍一看，似乎是一个很中规中矩的结构体

```
...
vari_struct  vari_1;
vari_struct* vari_p_1 = &vari_1;
vari_struct* vari_p_2 = malloc(sizeof(vari_struct))(
```

似乎都是这么用的，但总有那么一些人想出了一些奇怪的用法

```
int          what_spa_want = 10;
vari_struct* vari_p_3 = malloc(sizeof(vari_struct) + sizeof(long
)*what_spa_want);
```

这么做是什么意思呢？这叫做可变长结构体，即便我们超出了结构体范围，只要在分配空间内，就不算越界。`what_spa_want` 解释为你需要多大的空间，即在一个结构体大小之外还需要多少的空间，空间用来存储 `long` 类型，由于分配的内存是连续的，故可以直接使用数组 `vari_store` 直接索引。 而且由于C语言中，编译器并不对数组做越界检查，故对于一个有 `N` 个数的数组 `arr` ，表达式 `&arr[N]` 是被标准允许的行为，但是要记住 `arr[N]` 却是非法的。 这种用法并非是娱乐，而是成为了标准(**C99**)的一部分，运用到了实际中

对于内存的理解

在内存分配的过程中，我们使用 `malloc` 进行分配，用 `free` 进行释放，但这是我们理解中的分配与释放吗？ 在调用 `malloc` 时，该函数或使用 `brk()` 或使用 `mmap()` 向操作系统申请一片内存，在使用时分配给需要的地方，与之对应的是 `free` ，与我们硬盘删除东西一样，实际上:

```
int* value = malloc(sizeof(int)*5);
...
free(value);
printf("%d\n", value[0]);
```

代码中，为什么在 `free` 之后，我又继续使用这个内存呢？因为 `free` 只是将该内存标记上释放的标记，示意分配内存的函数，我可以使用，但并没有破坏当前内存中的内容，直到有操作对它进行写入。 这便引申出几个问题:

- Bug更加难以发现，让我们假设，如果我们有两个指针 `p1` , `p2` 指向同一个内存，如果我们对其中某一个指针使用了 `free(p1);` 操作，却忘记了还有另一个指针指向它，那这就会导致很严重的安全隐患，而且这个隐患十分难以发现，原因在于这个Bug并不会在当时显露出来，而是有可能在未来的某个时刻，不经意的让你的程序崩溃。
- 有可能会让某些问题更加简化，例如释放一个条条相连的链表域。

> > 某些大哥提到说， `free` 并不是什么都不做，而是将该段地址空间的前面一小部分置零 但是如果地址空间很长的话，依旧有误用的风险，希望大家还是警惕
>
> 实际上之所以库作者不让 `free` 操作将地址空间清空，有一部分原因是为了性能考虑，因为置零操作是一个消耗性能的行为，具体可以自行尝试，所谓双刃剑就在于此。

总的来说，还是那句话C语言是一把双刃剑。

0x06-C语言预处理器

预处理最大的标志便是大写，虽然这不是标准，但请你在使用的时候大写，为了自己，也为了后人。

预处理器在一般看来，用得最多的还是宏，这里总结一下预处理器的用法。

```
#include <stdio.h>
#define MACRO_OF_MINE
#ifdef MACRO_OF_MINE
#else
#endif
```

上述五个预处理是最常看见的，第一个代表着包含一个头文件，可以理解为没有它很多功能都无法使用，例如C语言并没有把输入输入纳入标准当中，而是使用库函数来提供，所以只有包含了 `stdio.h` 这个头文件，我们才能使用那些输入输出函数。 `#define` 则是使用频率第二高的预处理机制，广泛用在常量的定义，只不过它和 `const` 声明的常量有所区别:

```
#define MAR_VA 100
const int Con_va = 100;
...
/*定义两个数组*/
...
for(int i = 0;i < 10;++i)
{
    mar_arr[i] = MAR_VA;
    con_arr[i] = Con_va;
}
```

- 区别1，定义上 `MAR_VA` 可以用于数组维数，而 `Con_va` 则不行
- 区别2，在使用时，MAR_VA的原理是在文中找到所有使用本身的地方，用值替代，也就是说 `Con_va` 将只有一分真迹，而 `MAR_VA` 则会有 `n` 份真迹(n为使用的次数) 剩下三个则是在保护头文件中使用颇多。

几个比较实用的用于调试的宏,由C语言自带

- `__LINE__和__FILE__` 用于显示当前行号和当前文件名
- `__DATE__和__TIME__` 用于显示当前的日期和时间
- `__func__` **(C99)** 用于显示当前所在外层函数的名字

上述所说的五种宏直接当成值来使用即可。

- `__STDC__`
  - 如果你想检验你现在使用的编译器是否遵循ISO标准，用它，如果是他的值为1。

    ```
    printf("%d\n", __STDC__);
    ```

    `输出: 1`

  - 如果你想进一步确定编译器使用的标准版本是C99还是C89可以使用 `__STDC__VERSION__` ，C99(199901)

    ```
    printf("%d\n", __STDC_VERSION__);
    ```

    `输出: 199901`

> 可能很多人对这些宏没什么感触，实际上一般的确是用不到，但是：
>
> > 当你在写一些隐晦的东西时 `volatile int x = 10;`
> >
> > 你试试把这个代码用 `-std=c99` 编译一下，如果不出意外应该是出错的
> >
> > 在 ISO 标准里， `volatile` 是用 `__volatile__` 来实现的，这个对 `GCC` , `Clang` , `Visual C++` 而言都是如此 除此之外还有许多，有待你们自己发掘。

对于 **`#define`**

1. 预处理器一般只对同一行定义有效，但如果加上反斜杠，也能一直读取下去

   ```
   #define err(flag) \
       if(flag) \
         printf("Correctly")
   ```

可以看出来，并没有在末尾添加 `;` ，并不是因为宏不需要，而是因为，我们总是将宏近似当成函数在使用，而函数调用之后总是需要以 `;` 结尾，为了不造成混乱，于是在宏定义中我们默认不添加 `;` ，而在代码源文件中使用，防止定义混乱。

2. 预处理同样能够带来一些便利

```
#define SWAP1(a, b) (a += b, b = a - b, a -= b)
#define SWAP2(x, y) {x ^= y; y ^= x; x ^= y}
```

引用之前的例子，交换两数的宏写法可以有效避免函数开销，由于其是直接在调用处展开代码块，故其比拟直接嵌入的代码。但，偶尔还是会出现一些不和谐的错误，对于初学者来说:

```
int v1 = 10;
int v2 = 20;
SWAP1(v1, v2);
SWAP2(v1, v2);//报错
```

对于上述代码块的情况，为什么 `SWAP2` 报错？对于一般的初学者来说，经常忽略诸如 `，` `goto` `do...while` 等少见关键字用法，故很少见 `SWAP1` 的写法，大多集中于 `SWAP2` 的类似错误，错就错在 `{}` 代表的是一个代码块，不需要使用 `;` 来进行结尾，这便是宏最容易出错的地方 宏只是简单的将代码展开，而不会做任何处理 对于此，即便是老手也常有失足，有一种应用于单片机等地方的C语言写法可以在此借鉴用于保护代码:

```
#define SWAP3(x ,y) do{ \
        x ^= y; y ^= x; x ^= y; \
        }while(0)
```

如此便能在代码中安全使用花括号内的代码了，并且如之前所约定的那样，让宏的使用看起来像函数。

3. 但正所谓，假的总是假的，即使宏多么像函数，它依旧不是函数，如果真的把它当成函数，你会在某些时候错的摸不着头脑,还是一个经典的例子，比较大小:

```
#define CMP(x, y) (x > y ? x : y)
...
int x = 100, y = 200;
int result = CMP(x, y++);
printf("x = %d, y = %d, result = %d\n", x, y, result);
```

执行这部分代码，会输出什么呢？ 答案是，不知道！至少 `result` 的值我们无法确定，我们将代码展开得到

```
int result = (x > y++ ? x : y++);
```

看起来似乎就是 `y` 递增两次，最后 `result` 肯定是 `200` 。真是如此？C语言标准对于一个确定的程序语句中，一个对象只能被修改一次，超过一次那么结果是未定的，由编译器决定，除了三目操作符 `?:` 外，还有 `&&` , `||` 或是 `,` 之中，或者函数参数调用， `switch控制表达式，for里的控制语句` 由此可看出，宏的使用也是有风险的，所以虽然宏强大，但是依旧不能滥用。

4. 对于宏而言，前面说过，它只是进行简单的展开，这有时候也会带来一些问题:

```
#define MULTI(x, y) (x * y)
...
int x = 100, y = 200;
int result = MULTI(x+y, y);
```

看出来问题了吧？展开之后会变成: `int result = x+y * y;` 完全违背了当初我们设计时的想法，一个比较好的修改方法是对每个参数加上括号: `#define MULTI(x, y) ((x) * (y))` 如此，展开以后:

```
int result = ((x+y) * (y));
```

这样能在很大程度上解决一部分问题。

5. 如果对自己的宏十分自信，可以嵌套宏，即一个表达式中使用宏作为宏的参数，但是宏只展开这一级的宏，对于多级宏另有办法展开

```
int result = MULTI(MULTI(x, y), y);
```

展开成: `int result = ((((x) * (y))) * (y));`

> 实际上，并不要太追求用宏去替换函数，例如这个交换函数，老老实实写函数，有时候比宏更好

对宏的应用

1. 由于我们并不明白，在某些情况下宏是否被定义了，所以我们可以使用一些预处理保护机制来防止错误发生

   ```
   #ifndef MY_MACRO
   #define MY_MACRO 10000
   #endif
   ```

   如果定义了 `MY_MACRO` 那就不执行下面的语句，如果没定义那就执行。

2. 在宏的使用中有两个有用的操作符，姑且叫它操作符 `#` , `##`

   - 对于 `#` 我们可以认为 `#` 操作符的作用是将宏参数转化为字符串。

     ```
      #define HCMP(x, y) printf(#x" is equal to" #y" ? %d\n"
     , (x) == (y))
      ...
      int x = 100, y = 200;
      HCMP(x, y);
     ```

     展开以后

     ```
      printf("x is equal to y ? %d\n", (100) == (200));
     ```

     - 注：可以自行添加编译器选项，来查看宏展开之后的代码，具体可以查询 `GCC` 的展开选项，这里不再详述。特别是在多层宏的嵌套使用情况下，但是我不太推荐，故不做多介绍。

       - 能说的就是如何正确的处理一些嵌套使用，之所以不愿意多说也

不愿意多用，是因为C预处理器就是一个奇葩

- 举一个典型的例子， `__LINE__` 和 `__FILE__` 的使用。

```
  /* 下方会说到的 # 预处理指示器，这里先用，实在看不懂，
可以自己动手尝试 */
  #define WHERE_AM_I #__LINE__ " lines in " __FIL
E__
  ...
  fputs(WHERE_AM_I, stderr);
```

这样能工作吗？如果能我还讲干嘛。

```
  /* 常理上这应该能工作，但是编译器非说这错那错的 */
  /* 好在有前人踏过了坑，为我们留下了解决方案 */
   #define DEPAKEGE(X) #X
  #define PAKEGE(X) DEPAKEGE(X)
  #define WHERE_AM_I PAKEGE(__LINE__) " lines in
" __FILE__
  ...
  fputs(WHERE_AM_I, stderr);
```

不要问我为什么，因为我也不知道C预处理器的真正工作机制是什么。

第一次看见这种解决方案是在 **Windows** 核心编程 中，这本书现在还能给我许多帮助，虽然已经渐渐淡出了书架

总结起来，即将宏参数放于 `#` 操作符之后便由预处理器自动转换为字符串常量，转义也由预处理器自动完成，而不需要我们自行添加转义符号。

3. 对于 `##`
   它实现的是将本操作符两边的参数合并成为一个完整的标记，但需要注意的是，由于预处理器只负责展开，所以程序员必须自己保证这种标记的合法性，这里涉及到一些写法问题，都列出来

```
#define MERGE(x, y) have_define_ ## x + y
#define MERGE(x, y) have_define_##x + y
...
result = MERGE(1, 3);
```

这里首先说明，上述写法由于习惯原因，我使用第二种，但是无论哪种都无伤大雅，效果一样。上述代码展开以后是什么呢？

```
result = have_define_1 + 3;
```

在我看来，这就有点 `C++` 中模版的思想了，虽然十分原始，但是总是有了一个方向，凭借这种方法我们能够使用宏来进行相似却不同函数的调用，虽然我们可以使用函数指针数组来存储，但需要提前知晓有几个函数，并且如果要实现动态增长还需要消耗内存分配，但宏则不同。

```
    inline int func_0(int arg_1, int arg_2) { return arg_1
+ arg_2; }
    inline int func_1(int arg_1, int arg_2) { return arg_1
- arg_2; }
    inline int func_2(int arg_1, int arg_2) { return arg_1
* arg_2; }
    inline int func_3(int arg_1, int arg_2) { return arg_1
/ arg_2; }
    #define CALL(x, arg1, arg2) func_##x(arg1, arg2)
    ...
        printf("func_%d return %d\n",0 ,CALL(0, 2, 10));
        printf("func_%d return %d\n",1 ,CALL(1, 2, 10));
        printf("func_%d return %d\n",2 ,CALL(2, 2, 10));
        printf("func_%d return %d\n",3 ,CALL(3, 2, 10));
```

十分简便的一种用法，在我们增加减少函数时我们不必考虑如何找到这些函数只需要记下每个函数对应的编号即可，但还是那句话，不可滥用。

```
#define CAT(temp, i) (cat##i)
//...
for(int i = 0;i < 5;++i)
{
    int CAT(x,i) = i*i;
    printf("x%d = %d \n",i,CAT(x,i));
}
```

4. 对于宏，在使用时一定要注意，宏只能展开当前层的宏，如果你嵌套使用宏，即将宏当作宏的参数，那么将导致宏无法完全展开，即作为参数的宏只能传递名字给外部宏

```
#define WHERE(value_name, line) #value_name #line
...
puts(WHERE(x, __LINE__)); //x = 11
```

`输出: 11__LINE__`

5. 对于其他的预编译器指令，如: `#pragma, #line, #error` 和各类条件编译并不在此涉及，因为使用上并未有陷阱及难点。

6. **C**和**C++**混合编程的情况

    - 经常能在源代码中看见 `extern "C"` 这样的身影，这是做什么么的？
    - 这是为了混合编程而设计的，常出现在 **C++**的源代码中，目的是为了让 **C++**能够成功的调用 **C** 的标准或非标准函数。

    ```
    #if defined(__cplusplus) || defined(_cplusplus)
            extern "C" {
    #endif

            /**主体代码**/

    #if defined(__cplusplus) || defined(_cplusplus)
            }
    #endif
    ```

    这样就能在C++中调用C的代码了。

   - 在 **C** 中调用 **C++** 的函数需要注意，不能使用重载功能，否则会失败，原因详见C++对于重载函数的实现。也可以称为 **mangle**

7. 还有一种可以被称之为宏的小应用的技巧

   - 对于一个宏而言，是否有考虑过它的返回值是什么
   - 或者如何令其有一个函数那样的功能
   - 其实很简单

   ```
    #define TEST_RET(val, continues) ({continues = 19;val = 11;})
    ...
    {
        __attribute__((unused)) int oldval = 10;
        __attribute__((unused)) int newval = 18;
        fprintf (stderr, "%d\n", TEST_RET(oldval, newval));
    }
   ```

   - 可以尝试一下这个方法，其中原理自然就知道了。具体操作就是用 `({})` 包裹你想要的东西。

对宏的敬畏

1. 为什么有这么一说，因为使用宏真的是处处危险，而且代码难以调试
2. 经常会遇到这种情况，你将代码写成函数的时候没有任何问题，但是改成宏却出现了问题
   - 当然更可能的是你一开始就写宏，却发现总是得不到到预期的结果！
3. 不知道诸位对反转链表这种知识点掌握的如何？
   1. 如果很有信心不妨挑战一下下面的东西，看看是否能在我说出原由之前意识到问题
   2. 如果不太懂，那就跟着看下去，一定有收获！

> 举个例子最好说明问题

- 假设要写一个双向链表的插入操作
  - 我想要提供的是两个功能，后方插入，前方插入
  - 我的设计原型是 `Linux` 内核的链表原型。

> 所谓的 `Linux` 内核的链表原型 就是在内核编程中使用的链表数据结构，我以它为例子，自己写了一个插入操作

```
#define _list_add_inner(_add_pos, _add_node) \
do {\
    (_add_node)->next = (_add_pos)->next;\
    (_add_node)->prev = (_add_pos);\
    (_add_pos)->next->prev = (_add_node);\
    (_add_pos)->next = (_add_node);\
} while(0)

static inline void list_add_after(struct list * add_pos, struct
list * add_node) {
    _list_add_inner(add_pos, add_node);
}

static inline void list_add_before(struct list * add_pos, struct
 list * add_node) {
    _list_add_inner(add_pos->prev, add_node);
}
```

- 很好，可以试着测试一下最后这两个函数 `list_add_after` ， `list_add_before` 看看是否达到预期目的？

> 有时候代码真的就是要测试才行

- 不啰嗦，这样是不行的！
    - 为何？问题就出在 `list_add_before` 这个函数的 `add_pos->prev` 参数上，原因就是宏只是做一个简单的替换，而不是值代入
    - 这里需要自己体会一下。修正一下代码

> 替换和值代入可是大不相同的

```
#define _list_add_inner(_add_pos, _add_node) \
do {\
    struct list * tmp = _add_pos;\
    (_add_node)->next = tmp->next;\
    (_add_node)->prev = tmp;\
    tmp->next->prev = (_add_node);\
    tmp->next = (_add_node);
} while(0)
```

- 不知是否看出了什么门道，这就是关键所在，构造一个值，而不是简单的替换。可以自己动手画一画流程图。

0x07-C语言效率(上)

大概所有学习C语言的初学者，都被前辈说过，C语言是世界上接近最速的编程语言，当然这并不是吹牛，也并不是贬低其他语言，诚然非C语言能写出高速度的代码，但是C语言更容易写出高速的程序(高速不代表高效)，然而再好的工具，在外行人手中也只能是黯淡没落。

对于现代编译器，现代CPU而言，我们要尽量迎合CPU的设计(比如架构和处理指令的方式等)，虽然编译器是为程序员服务，并且在尽它最大的能力来优化程序员写出的代码，但是毕竟它还没有脱离电子的范畴，如果我们的代码不能让编译器理解，编译器无法帮我们优化代码，那么我们就无法写出一个高速的程序。

对于此，我们可以暂且忽略CPU的设计，因为我们在层面上只能考虑如何迎合编译器的优化规则，而CPU则是语言以及编译器的事情了。

提高程序的速度，就C语言而言可以有这几种方法:

- 首先还是要设计合理的大纲，正所谓一个程序最大的性能提升就是它第一次运行的时候
- 要避免连续的函数调用。
- 消除不必要的存储器使用(并非推荐使用register)
- 使用循环展开技巧，一般编译器的优化选项能自动帮你修改代码成循环展开
- 对于一个操作的核心耗时部分，通过重新组合技术来提高速度
- 多采用几种风格的写法，而不是直观的认为，因为计算机的想法和你是不一样的
- 注：随着编译器的版本更新，即使不开启优化选项，自带的编译器优化依旧能够为我们编写的代码提供一部分优化，这便是不使用老版本编译器的原因，虽然作为一个程序员不应该太依赖于编译器，但是我认为，时代在进步，信息量正在无限的膨胀，但是人类的大脑以及精力在一个大时代内是有限的，换句话说对于普通人而言我们的记忆是有限的，我们不应该把精力放在前人已经做完的事情上，而是要站在巨人的肩膀上向更远处眺望，如此我们应该充分利用工具来帮助我们实现一些既有的功能，而程序员应该更 专注于发现新的思路，以及想法，在图灵测试尚未有人打破之前，程序员依赖编译器并不是一件错误的事情。

  对于当下的编译器，以 `GCC` (**GCC**不仅仅是一个编译器，但这里将它当成编译器的代名词)为例， `-O2` 是一个为大众所接受的优化等级，对于其他编译器，一般程序员可以选择使用由Google和Apple联合开发的编译器 `clang` 也是一

个很好的选择， 在 `-O2` 的优化等级下， `GCC` 一般情况下能够自动执行循环展开优化，

开始

1. .

```
/*struct.h*/
#include <stdio.h>
typedef struct me{
        int         value;
        struct me* next;
}data_t;

typedef struct{
        int index;
        data_t* storage;
}block;
```

为了测试方便我们首先定义了两个结构体，分别是:

`block` 代表一个块，每个块都有一个序号(`int`)，一个数据域 `data_t`
`data_t` 代表一个数据域，原型是一个链表，每个 `data_t` 对象中包含一个数据和一个指针。

```
/*main.c*/
#include "struct.h"
#define ARR_SIZE 10
static inline int get_len(const data_t* data)
{
    int len = 0;

    if(!data)
        fprintf(stderr,"The data in %p is NULL\n",data);
    else
        while(!data->next)
        {
            ++len;
            data = data->next;
        }
    return len;
}

static inline void mix_cal(const block* process, int result
[])
{
    for(int i = 0;i < get_len(process->storage);++i)
    {
        *result += (process->storage)[i];
    }
}
```

此时我们得到了两个测试函数，`get_len` 和 `mix_cal` 分别用来得到 `data_t` 长度，以及计算数据域的总和。

```
/*main.c*/
int main(void)
{
    block* block_in_all[ARR_SIZE]  = { NULL };
    int    result_in_all[ARR_SIZE] = { 0 };
    /*
     * 假设生成了许多的`block`类型对象
     * 将许多的`block`放置在一个数组中，每个元素类型为`block*`
     * 每个block对象中都包含非空的data_t类型的数据域
     */
    for(int i = 0;i < ARR_SIZE;++i)
    {
        mix_cal(block_in_all[i], result_in_all+i);
    }
    for(int i = 0;i < ARR_SIZE;++i)
    {
        printf("The %dth block have the total %d data\n",
                    block_in_all[i]->index, result_in_all[i
]);
    }

    return 0;
}
```

耐心读完上述的代码，它是用来求和的，求一个域中的所有元素的和。仔细分析一下，很容易就能看见一些缺点，最大的莫过于在 `mix_cal` 函数中对于 `get_len` 函数的调用，在此处看来十分明显，但是我们在编写程序的时候是否能够注意到这个问题呢？
对于一些不必要的函数调用我们要做的便是将他们提取出来，使用临时变量是一个很好的办法，因为在编译器的帮助下临时变量在允许的情况下能够充分的利用CPU的寄存器。之所以是允许的情况下，是因为寄存器的数量并不多，而编译器在寄存器的使用上需要考虑许多的复杂因素，故并不是每次使用临时变量都能加入寄存器。但这并不妨碍我们提升程序的性能。

在此处，我们应该将 `for` 循环中的判断语句里的 `get_len` 函数提取出来，在外部使用一个临时变量接收结果，而不是在循环中一直调用该函数。

```
int len = get_len(process->storage);
```

2. .

依旧是上方的代码，我们来讲述一下，循环展开。

对于 `mix_cal` 函数，我们或者说编译器可以如何提升它的速度呢？我们说过一点的小改变都可能对一个程序的最终代码产生极大的影响，对此最常用的便是尝试，前人之路早已铺好，不需要重复造轮子了。

循环展开:

```
int reality = len - 1, i;
for(i = 0;i < reality;i+=2)
{
    *result = *result + (process->storage)[i]
                      + (process->storage)[i+1];
}
for(;i < len;++i)
{
    *result +=  (process->storage)[i];
}
```

这就是循环展开中的**2**次循环展开，同样还有n次循环展开。

同样，在刚才提到过寄存器的使用以及减少不必要的开销，在此程序中对于 `(process->storage)[i]` 这样的存储器位置解引用太过浪费，我们总是将其优化成本低临时变量的使用

```
data* local_data = process->storage;
```

这将为程序带来十分可观的节约，虽然这些工作在编译器的优化中都能包括，但是一旦我们的代码难以被编译器所理解(虽然编译器的升级最大的目的就是提升优化效果)，那么我们很可能得到一个性能不够可观的程序。所以当我们并不是特别紧凑的时候，可以将这些工作当成我们的本分来做，而不是交给编译器来做。

以及对于外部存储位置 `result` 我们在此处也是存在着浪费，同样我们应该使用一个临时变量来存储总和，而不是每次得到结果便对它进行解引用操作。

```
int local_result = 0;
/*...*/
local_result = local_result + local_data[i] + local_data[i+
1];
/*...*/
*result = local_result;
```

在上方我们可以看见循环展开被称作**2**次循环展开，那么自然可以推断有 `n` 次循环展开，自然是有的，对于一个n次循环展开的式子我们有一个简便的上界确定公式即:

```
reality = len - n + 1;
```

至于展开几次最好，依然是视环境而定。 故最终的版本应该为：

```
static inline void mix_cal(const block* process, int result
[])
{
    int local_result = 0;
    int len = get_len(process->storage);
    int reality = len - 1, i;
    data* local_data = process->storage;

    for(i = 0;i < reality;i+=2)
        local_result += local_data[i] + local_data[i+1];
    for(;i < len;++i)
        local_result += local_data[i];

    *result = local_result;
}
```

解释：循环展开将元素相加分为两个部分，第一部分每次加两个元素，由于如此做会剩余元素没有加，故在第二部分将剩下的元素都加起来。

3. . 还有一种叫做重新组合的技巧，即为让一个表达式中的运算数自由组合，组合出最快速的一种，但是这种方法未曾试验过。故不提及。

4. . 对于条件分支预测错误造成的时间损耗，称之为惩罚，最通俗的说法，就是当你编写的代码中含有条件分支的时候，处理器会选择去预判某一个分支是此次正确的支路，这样可以避免修改任何实际的寄存器和存储器，一直到确定了实际结果，要是不对，那就惨了，这段时间做的事情都白费了。但是也不必过分的关心这种条件分支的预测，这也是我放在最后说的意义所在。

   这里有两种较为客观的方法，一种被称为命令式，一种被称为功能式

   命令式：

```
for(int i = 0;i < n;++i)
{
    if(a[i] > b[i]){
        int temp = a[i];
        a[i] = b[i];
        b[i] = temp;
    }
}
```

   功能式：

```
int min, max;
for(int i = 0;i < n;++i)
{
    min = a[i] < b[i] ? a[i] : b[i];
    max = a[i] < b[i] ? b[i] : a[i];
    a[i] = min;
    b[i] = max;
}
```

   很清晰的一个例子，明显看出来，前者对于不同情况所作的程序步数明显不同，而后者无论什么情况都是相同的程序步。

   两个形式的好处前者对于可预测数据来说，是一个很好的模型，后者则是中庸之道，什么是可预测不可预测，比如一个数是负数还是正数这就是不可预测的，用前面那个代码会有很大的惩罚。

5. . 多路并行的技巧也是一个很重要的思路，可能在很多人眼中看来，两条语句依次写出和合并的效果一定是一样。但是多路并行有一个缺点就是对寄存器的数量有所要求，当寄存器不够时(称为溢出)，性能不升反降。同样是对于循环展开，此次使用四次循环展开加二路并行：

```
for(i = 0;i < reality;i+=4){
    local_result_1 += local_data[i] + local_data[i+1];
    local_result_2 += local_data[i+2] + local_data[i+3];
}//也可以分成四路并行，每一路存一个。这种做法充分利用了CPU流水线的性能
for(;i < len;++i)
    local_result_1 += local_data[i];

*result = local_result_1 + local_result_2;
```

结束

Tips:

上文中写到的函数大都带有 `static inline` 关键字，这是何意？首先我们要确定一件事情，对于非工程的单文件而言， `static` 函数并没有什么意义(意义指的是对于可见性而言，并非说它一无是处)，许多人对于 `static` 函数感到茫然的原因在于:我明明将一个函数声明定义成 `static` 类型了，但是我还是可以在别的文件中访问到啊！

其实这是因为你根本就没有理解**C**语言工程这个意思，大部分人是这么测试的:

1. 首先在一个文件夹里创建两个文件 `test_static.c` 和 `static.h` :

```
/*static.h*/
#ifndef STATIC_H
#define STATIC_H
static void test(void);

static void test(void);
{
    printf("Hello World!\n");
}
#endif
```

...

```
/*test_static.c*/
#include <stdio.h>
#include "static.h"

void test(void);
int main(void)
{
    test();         //编译通过，可以运行。
    return 0;
}
```

2. 然后编译运行，发现可以通过啊！！标准怎么说在其他文件中不可见？而把 `static.h` 去掉 `#include` 之后发现报错 `test undefined` ，瞬间初学者就凌乱了。

3. 好吧，实际上是前辈们以及教材的错，因为从始至终，所有外界现象都告诉我们**C**程序是独立的一个一个文件组成的，但是并没有告诉我们要先将他们弄成一个工程！此处如果是使用**Visual Studio**学习C语言的可能会对工程这个概念理解的稍微好一些，虽然不推荐使用 VS 学习C语言。

4. 你想要实现 `static` 函数仅在本文件可见的效果，请你先补习一下工程这个概念，对于任何可见或者不可见的概念而言都是建立在一个工程内而言，而不是像上方的代码，使用 `#include` 来表示，你都 `#include` 了，那还有什么可见不可见的当然都可见了。所以一个 `static` 函数可见于不可见是基于一个个工程里的所有C语言源文件而言的。所以你将常看见前辈们这么回答你的提问:

```
/*static.h*/
#ifndef STATIC_H
#define STATIC_H
static void test(void);

static void test(void);
{
    printf("Hello World!\n");
}
#endif
```

...

```
/*test_static.c*/
#include <stdio.h>

void test(void);
int main(void)
{
    test();         //报错，因为test是static函数。
    return 0;
}
```

发现了吗？在上方代码中，少了一行 `#include "static.h"` 但是这个代码依旧可行，因为这两个文件是建立在同一个工程里的，而不是在一个文件夹中随意新建两个源文件这么简单，你可以使用各个**IDE**的工程功能来进行测试。

回到正题，在这里稍微提一下**static**对函数的某些作用，它可以让函数放在一个静态的空间中，而不是栈里，这是的它的调用更加快速，经常与**inline**关键字一起使用，为的就是让函数更加快。但是有利有弊，可以自己权衡一下。

参考:深入理解计算机系统--Randal E.Bryant / David O'Hallaro

0x08-C语言效率(下)

注：存储器山就是对于不同步长不同大小文件的读取速率的三维坐标图，形似一座山，`z` 轴为速率，`x` 轴为步长，`y` 轴为文件大小（字节），某些主流的测评软件便是这个原理(将存储器山的图像进行一下简单的变换，就能得到哪些软件呈现的效果图像)。

上文提到过，任何一点小改动，都有可能让程序的性能发生很大的变动，这是为什么？

当时我们并未深究，由于我们惯性的认为计算机的运作方式和人类的运作方式一致，也在过往的经验中认为计算机一定是在任何方面超越人类的存在，但是实际上，计算机除了在重复计算方面比人类的速度要快速以外，其他方面远远落后于人类的大脑，即便是我们最稀疏平常的视觉识别(看东西识别物体)，在计算机看来都是一门极其高深的领域，所以我们现在的时代的计算机还处于起步状态，在这种时代里，程序员的作用是无可替代的，同样程序员的一举一动关乎计算机的命运。

可能在很多的方面，都已经接触了一台计算机的主要组成构造，和程序员最息息相关的便是CPU，主存以及硬盘了，可能到现在为止很多程序员仍然认为编程序和这些存储器有什么关系？然而一个程序员，特别是编写C语言程序的程序员，最大的影响因素便来自于此，在计算机的存储器结构中，分为四种层次:
CPU寄存器 高速缓存器 主存 硬盘

但是有没有想过，为什么计算机存储器系统要分成这四层结构呢？我们知道，上述四种存储器的读写速度依次降低，我们为什么不选择一种速度中庸的，价格也中庸的材料，制造所有层次的存储器呢？

- 有人给出的解释是，一个编写良好的程序总是倾向于访问层次更高的存储器，而对于现在的技术，价格高昂而无法大量制造的高速存储器来说，我们可以选择按层次分配构造，让我们以最低的成本的存储器达到使用最高的速度存储器的效果。
- 就像是在自己的计算机上，当我们打开一个很笨重的应用程序后，会发现，下一次再打开的时候可能会更快，就像以前历史遗留的一个问题 **Visual Studio 2008** 在 **Windows XP** 上，第一次打开总是十分卡顿，但是当关闭程序之后第二次打开却是很流畅。在参考书中，提到过两个评价程序速度的关键点：时间局部性和空间局部性 。
    - 时间局部性：在访问过某块存储器之后的不久的将来，很可能会再次访问

    它
  - 空间局部性：在访问过某块存储器之后的不就的将来，很可能访问其邻近的存储器位置。
  - 良好的局部性改进一般能很好的提升程序的性能。
- 所谓局部性就是当我们使用过某些资源后，这些资源总是以一种形式存储在更高级更方便的存储器当中，让最近一次的存取请求能够更加有效率的进行。
  - 打个不太贴切的比喻，假设计算机是一个家，CPU是一个人，想象一下这个家中的所有物品都是井然有序的，这个人想要工作必然会需要工作物品，所以他需要从某些地方拿来，用完以后再放回去，这些地方就是存储器，但是过了一段时间发现这么做太浪费时间，有时候某些东西太远了，所以，人想把它把它放在离自己更进的地方，这样自己的效率就高很多，如果这个东西一段时间内不再用，则把它放回原处，留出位置给更需要的工作物品，于是形成了越常使用的物品离人越近的现象。这便是计算机存储器的分层结构的意义。
  - 而对于一个有良好局部性的程序而言，我们总能在离自己最近的地方找到我们所需要的数据，回到计算机：我们知道计算机的存储器是分层结构的，即每一层对应着不同的读写速度等级(CPU寄存器 > 高速缓存 > 主存 > 硬盘)，而我们的程序总是按照从左至右的顺序依次查找，每次找到一个所需要数据，不出意外，总是将其移动到上一层次的存储器中存储，以便下次更高速的访问，我们称这种行为叫做 命中 。越好的程序，越能将当时所需的数据放在越靠近左边的地方。这便是局部性的意义所在。
  - 当然，存储器如此分层也是出于无奈，在处理器的速度和存储器的速度实在差距的情况下只有如此做才能让处理器更加充分的利用，而不至于等待存储器读写而空闲，也许某一天，当内存的位价和普通硬盘不相上下或者差距不多的时候，也许内存就是硬盘了。而当今也有人使用某些特殊的软件在实现这个功能，凭着自己计算机上大容量的内存，分割出来当作硬盘使用，存取速度让硬盘望尘莫及。

局部性

前方提到了局部性，局部性体现在了，当步长越大，空间局部性越低，大多数情况下会造成性能降低，比如最常见的多维数组循环(我鲜少使用多维数组的原因之一便在于此)，前面说过多维数组实际上只是数个一维数组的包装而已，C语言中并没有眞正的多维数组，而是将其解读成内存中的一维的连续内存，但是当我们遍历它的时候，C语言为了不让我们被底层实现所困扰，所以生成了多维数组遍历的假象：

让我们重温一遍"多维数组"：

```
#include <stdio.h>
int main(void)
{
    int dim_1_arr[4]    = {1, 2, 3, 4};
    int dim_2_arr[2][2] = { {1, 2}, {3, 4} };
    int result_1 = 0;
    int result_2 = 0;

    for(int i = 0;i < 4;++i)
        result_1 += dim_1_arr[i];
    return 0;
}
```

此例中，对一维数组进行步长为 `1` 遍历求和，假设内存中数组的起始位置是 `0`

`0 => 4 => 8 => 12`

```
for(int j = 0;j < 3;++j){
    for(int i = 0;i < 3;++i){
        result_2 += dim_2_arr[i][j];
    }
}
```

此例中，我们的步长是多少呢？我们来看一下

`0 => 8 => 4 => 12`

可以很清晰的看出两段不同代码之间的跳跃，为什么？观察到多维数组的遍历中我们和平时的做法有些不同，是先对 `i` 进行遍历，再对 `j` 进行遍历，这就导致了程序必须在内存块中无规律的跳动，这里的无规律是计算机认为的无规律，虽然在我们看来的确是有迹可寻，优秀的编译器能够对它进行优化处理。就事论事，即这段程序的空间局部性比较差，对于一个在内存中大幅度跳跃，无规律跳跃的程序都将影响程序的性能。这个判定对于一个连续的内存块来说是很重要的，比如C语言中的结构体。

实际上C语言也是能够面向对象的，但是十分复杂，就像拿着棒子织衣服一样。而C语言的机构体能够让我们在一定程度上初步理解对象这个概念，因为它是一个完整的个体，虽然对外界毫不设防。

> 对于结构体

```
#define VECTOR 4
typedef struct{
        double salary;
        int    index[4];
}test_data;

int main(void)
{
    int result_1 = 0;
    int result_2 = 0;
    test_data dim_1_arr[VECTOR];
    /* ...填充数据 */

    for(int i = 0;i < VECTOR;++i)
    {
        for(int j = 0;j < 4;++j)
            result_1 += dim_1_arr[i].index[j];
    }/* for loop 1 */

    for(int j = 0;j < 4;++j)
    {
        for(int i = 0;i < VECTOR;++i)
            result_2 += dim_1_arr[i].index[j];
    }/* for loop 2 */
    return 0;
}
```

还是和上方一样，假设 `dim_1_arr` 起始位置为 `0`

for loop 1：

`8 => 12 => 16 => 20 ==> 32 => 36 => 40 => 44 ==> ...`

for loop 2：

`8 => 32 => 56 => 80 ==> 12 => 36 => 60 => 84 ==> ...`

从上方不完整的比较来看，**loop 1** 相对于 **loop 2** 来说有更好的空间局部性，很明显在 **loop 2** 中，CPU读取是在无规律的内存位置跳跃，而 **loop 1** 则是以单调递增的趋势向前(这里的向前指的是直观上的向前)读取内存。

- 在此处回顾一下C语言的结构体性质与知识：
  - 对于任意一个完整定义的结构体，每一个对象所占有的内存大小都符合内存对齐的规则。
  - 对于结构体内的各个成员而言，其相对于对象存储地址起始的距离，称为偏移量。
- 解释：
  - 内存对齐便是对于一个结构体而言，其所占内存大小总是最大成员的整数倍，其中最大成员指的是最基本成员，即：

```
typedef struct{
    test_data test_1;
    int       test_2;
}test_data_2;

/*...*/
printf("The size of test_data_2 = %d\n",sizeof(test_data_2));
/*...*/
```

输出： `The size of test_data_2 = 32`

```
typedef struct{
        int index[4];
        int store_1;
        int store_2;
}test_data_3;
typedef struct{
        test_data_3 test_3;
        int         test_4;
}test_data_4;

/*...*/
printf("The size of test_data_4 = %d\n",sizeof(test_dat
a_4));
/*...*/
```

`输出： The size of test_data_2 = 28`

仔细对比 `test_data_3` 与 `test_data` 的差异，可以发现不同处，在前者的内部包含了一个 `double` 类型的成员，在我的机器上它的长度为 `8`，后者的内部包含了两个 `int` 类型的成员，每个长度为 `4`，但是他们的长度在直观上是一样的！但是真正在使用的时候我们才能察觉到其中的差异，这就是我所说的最基本成员的意义所在。虽然我们在使用结构体的时候，能够将其当作一个整体，但是实际上他们与内建(build-in)的类型还是有一些差异的。

- 偏移量通俗地说，就是该成员起始地址距离起始位置的长度，在结构体中，C语言是怎么为结构体分配设定大小的呢？除了内存对齐外，还需要考虑定义结构体时，其中成员的声明顺序，换句话说，谁首先声明，谁的位置就靠前。而某个成员的偏移量代表着其起始位置减去其所属对象的起始位置，(此处需要注意的是，两个毫不相干的指针相减所得到的结果是无意义的，只有当两个指针同在一个作用域内时，减法才是有意义的，为了避免潜在的错误，我们要谨慎使用指针减法操作)。

- 就此回过头去再看看上方的 **loop** 解释，应该能够理解到，那些数字是通过偏移量来进行计算得到的。

- 之所以没有详细的介绍时间局部性是因为，对于时间局部性而言，其最大的影响因素便是操作区域的大小，比如我们操作的数组或者文件的大小，越小时间局部性越好，试想一下对于一个小的文件和大的文件，我们更容易操作到同一

块地方多次的，必定是小的文件。而操作文件的大小有时候并不能很好得成为我们的操作因素，故只能多关注空间局部性。

高速缓存器

1. 在前方提到了，一般情况下，局部性好的程序能够让程序比局部性差的程序更有效率，而对于局部变量而言，一个好的编译器总是尽可能的将之优化，使其能充分使用**CPU**寄存器,那么寄存器的下方,也就是速度最接近寄存器的,便是所谓的高速缓存器了，对于高速缓存器而言，其最大的功效便是缓冲，缓冲有两层意思：

    - 缓存数据，使下一次需要的数据尽可能的“靠近”CPU，此处的靠近并不是物理意义上的距离靠近。
    - 缓冲一下CPU于存储器巨大的速度差距，防止CPU空闲浪费。

2. 对于现在的计算机而言，CPU基本都是三层缓存：一级缓存**(L1)**,二级缓存**(L2)**,三级缓存**(L3)**，可以通过 **CPU-Z(Windows) / Mac OS**系统报告 来查看自己的CPU缓存，在软件中我们能够看到，在一级缓存中会分为两个部分 ：一级数据，一级指令，这代表着只读写数据，只读写指令，这样分开的意义在于处理器能够同时处理一个数据和一个指令，上述所说的都是对于一个CPU核而言的，也就是说当CPU是多核的时候，那就有多个这种**“**功能集合**(L1+L2)”**。二级缓存则与一级缓存同在一个核中，每个核都拥有自己的二级缓存，最后所有核共享唯一一个(L3)

    - 总的来说，对于高速缓存器来说，一般分为三层，第一层比较特殊由独立的两个部分组成，第二层第三层则是各自独立一体并未区分功能(既存数据又存指令)，而第一层和第二层则是每个核单独享有不同的缓存器，第三层则是各个核共享一个层，所以我们经常看见在个人计算机上，L3的大小经常是以**MB**为单位的，而第一层则多以KB甚至是Byte为单位。
    - 在实际中，喜欢研究计算机的人经常会在一些专业软件中看见自己的**CPU**配置，在缓存一栏的一级和二级中总能看见 `2 x 32 KBytes` 之类的参数， `32` 代表的就是某级的缓存大小，而前方的 `2` 则是核数，即有几个核便有乘多少，和之前所说的一致，具体可参见下方的缓存器图示

1. 高速缓存器的各个层依然遵守逐步降速的规律，即读取周期 **L1 < L2 < L3**，而影响较大的便是上文提到的的命中率，我们知道越上层的高速缓存器总是将下层的存储器映射在自己的存储器中，而按照逻辑推断，上层的实际空间比下层的要小，因为上层的空间更加宝贵速度更快，这就导致我们无法将下层的空间

一一对应的映射到上层里，那么我们就想到一个办法，并不是将下层存储器的内容完全映射到上层，而是上层有选择性的将下层的部分内容抽取到上层，这便是不命中之后的操作。

2. 对于CPU从存储器中读取数据这个操作，如果我们使用了高速缓存以及内存这两个概念，那么就会有一个延伸概念，命中。而对于这个概念只有两种情况，命中或者不命中。而对于一个初始化的高速缓存器，它一定是空的，也许在物理意义上它并不是空，但是实际上在程序看来它的确是空的，为了区分这个，高速缓存器专门使用了一个位**(bit)**来表示此组是否有效(即是否为空)，既然它是空的那么，我们第一次无论如何都无法命中数据，这时候该层的高速缓存器就会向下一层，在该层中寻找所要的数据，每次要向下一层申请寻找的行为一般称为惩罚，而当我们从存储器中将所需的数据加载到高速缓存器中的时候，我们便开始了运算，而一切关于高速缓存器效率的改进都集中在命中率的提升。

    - 假设有一个数组需要操作，由于数组是一个连续的内存空间，对其进行步长为 `1` 的操作拥有很好的空间局部性，那么可以当成一个很好的例子，在高速缓存器看来读取一个有 `n(n>N)` 个元素的数组 `vector` 并不是一次性读完，而是分次读取，如果读取了 `k` 次那么至少有 `k` 次不命中，这是不可避免的，而对于读取的数据也不一定是我们需要的，用书上的例子来说：
      `vector:|[0]|[1]|[2]|[3]|[]|[]|[]|[]|[]|[]|[]|`
      假设操作数组的每一个元素，我们一次读取三个内存的值，类型为 `int` ，因为原理都一样。那么在初始化时候，高速缓存器为空，在第一次操作的时候，读取了四个(如上所示)，此时一定经过了一次 不命中 。

      很好理解，因为缓存器空，所以第一次操作必然不命中，所以我们需要向下级存储器读取我们需要的数据，那么第二访问高速缓存的时候，可以命中 `vector[0]` ，依次命中后续两个，直到需要 `vector[4]` ，出现了不命中，那么我们就需要重复上一步，再次读取三个数据，依次类推直到结束。
      `vector:|[0]|[1]|[2]|[3]|[4]|[5]|[6]|[7]|[]|[]|[]|`

      现在我们能够从一定层面上解释为什么局部性好的程序比局部性差的程序要有更好的效率了，原因就在对于高速缓存器的利用，首先反复利用本地临时变量能够充分的调用高速缓存器的功能做到读写的最优化，其次步长为越小也越能尽最大的能力发挥高速缓存器读取的数据，在这点上再回过

头思考多维数组的遍历并进行操作，如果没有考虑空间局部性(即先操作大块，再操作小块)，那么在高速缓存器中，它的不命中率令人发指，这也是操作不当效率低的原因。

- 另一方面，对于不同步长而言，其影响的也是高速缓存器的命中率，还是上方的 `vector`

```
步长        | 1 | 2 | 3 | 4 | 5 |
不命中/命中 |1/4|1/2|2/3|1/1|1/1|
```

  可以看出来，对于步长而言，当到了一定的上限以后，每次的请求都会不命中，那么这时候本层的高速缓存器相当于作废，时间全都耗费在下层数据传送到上层的时间，因为每次读取都是不命中，可以利用上方的例子自己试着推理一下。

- 以上所说的每次读取下一级别存储器数据的时候，都是按照内存对齐，来读取的，如果你的内存数据，例如读取结构体时，没有放在内存对齐的位置(此处所说的内存对齐位置是以每次该级别存储器读取的大小为对齐倍数，而不是结构体在内存中存储时的内存对齐位置)，那么会将该位置的前后补齐倍数的起始位置来读取

```
下一级别存储器       0 1 2 3 4 5 6 7 8 9 A B C D E F
结构体数据存放位置在 4~F
每次该级别的存储器读取 12个数据
那么本次由于结构体存放的没有对齐到提取的内存位置，所有提取的可能
会是 0~B
```

  也就意味着，并不是每次缓存读取都是如此的完美，恰好都从内存中数据的首部开始读取，而是整片根据内存倍数进行读取。

3. 在参考文献中提到了一种优化程序的技巧，便是充分的利用高速缓存器，并且不受缓存器大小的限制，做法是当所操作的数据过大的情况下，通过构造循环来创建一个有一个大块，这些块能够被高速缓存器容纳，那么我们就能够充分利用高速缓存器来实现功能。

> 缓存器示意图



参考:[1]深入理解计算机系统--Randal E.Bryant / David O'Hallaro

0x09-未曾领略的新风景

- 前方曾提到两个关键字 `restrict` 和 `inline` 在C语言中的使用，但是后者可能还能带来些许理解上的便利，开启 `-O3` 优化是一个很不错的选择。
- `inline` 的作用还是在于和 `static` 一起使用，让小函数尽可能的减小开销甚至消除函数开销。
- `restrict` 最重要的还是在于编译器的优化上。编译器能够为我们的程序提供优化，这是众所周知的，但是编译器是如何优化的，知道的人少之又少，其中有一些优化是建立在编译器能够理解你的代码，或者说编译器要认为你的代码是可以被优化的情况下，才会采取优化措施：

    - 有一个很重要的地方，称为指针别名，是阻碍编译器优化代码的最重要的地方
    - 什么是指针别名？

    ```
    void tmp_plus(int * a, int * b)
    {
        for(int i = 0; i < b_len;++i)
            *a += b[i];
    }
    ```

    这段代码中， `a` , `b` 是两个被传入的指针，编译器对他们毫无所知，也不知道 `a` 是否在 `b` 的范围之内，故无法对其做出最大程度上的优化，这会导致什么结果呢？也就是，每依次循环过后， `*a` 的结果都会写回到主存当中去，而不是在寄存器里迅速进行下一次增加！

    或者有的聪明的编译器可以将其扩展成 `if ... else` 的加长版形式来避免写回操作。

    但是如果我们增加了 `restrict`

    ```
    void tmp_plus(int * restrict a, int * restrict b) ...
    ```

    这就是告诉编译器，这两个指针是完全不相干的，你可以放心的优化，不会出错。

- 但是在这里有一些小的问题，那就是 `C++` 并不支持这个关键字，这会导致什么后果？

    - 你在 `Visual Studio` 下编程的时候会发现使用 `restrict` 关键字是会产生编译错误的，无论你使用 `.c` 还是 `.cpp` ，难道说不支持吗？实际上不是，主流的编译器都对这个关键字有自己的实现
    - **Visual Studio(Visual C++)**： `__restrict`
    - **GCC**, **Clang**： `__restrict__`
- 剩下一个是前面也大概说过的 `volatile` ，当时对其的解释就是让编译器不对其进行优化的意思，这里再说清楚一点

    - 假设 `volatile int i = 0;`
    - 首先它的现象本质就是，确保每次读取 `i` 的时候，是从它的内存位置读取，每次对它操作完毕后，将结果写回它的内存位置，而不是将其优化保存在寄存器内。
    - 这就让一些编译器的优化无法进行，就像上方所说的。
    - 一般将其用在调试时期，防止编译器的优化对自己的代码逻辑造成混淆。
    - 但是，正如上面所说，这个关键字的作用是每次都进行存取，开销自然就变大了，意味着无法使用缓存来对其进行加速，换句话来说就是，只要是关于它的操作，开销都将变大。
    - 并且，其所能起到的作用大部分体现在 多线程编程中，而且也无法阻止指令重排之类的优化。
        - 对此，有一个需要提及的内容是，可以适当的使用 内存屏障 来替代这种 `volatile` 的功能，内存屏障是由操作系统提供的功能，目的是防止由于某些优化，导致的指令重排的效果。
        - 某些编译器也有提供类似的功能，例如 **GCC**就可以通过内嵌汇编代码的方式实现这个效果
    - 以上的略微提及，详细可以自行查阅资料。

再议数组

- 在常见C中，数组是这样的。

```
int arr_1[3];
int arr_2[] = {1, 2, 3}; /* 创建三个元素的数组 */
```

- **C99**之后，可以使用一种叫做 复合文字**(Compound Literal)**的机制来做到更多的事情，最简单的就是创建匿名数组(看着有点像C++11引进的 **Lambda**匿名函数)：

```
int *ptoarr = (int[]){1, 2, 4}; /* 之后可以使用 ptoarr 操作 */
ptoarr[2] = 0;
printf("The Third number is : %d", ptoarr[2]);
```

输出： `$ The Third number is : 0`

当然，这种机制并不是只能如此使用，稍微高级一点的应用是，可以传递数组了，无论是按参数传递还是返回值。

```
int *test_fun(int most[], int length){
    for(int i = 0;i < length;++i)
     most[i] = i;
return (int []){most[0], most[1], most[2], most[3]...};/* so on */
}
// main
test_fun((int []){6,6,6,6,6}, 5);
```

这也是自从更新了C99标准以后，可以讲某个整体进行返回的例子，也包括结构体：

```
    typedef struct compond{
            int value;
            int number;
            int arrays[10];
    }compond;
    //假设有test_fun函数返回该结构体
    ...
    return (combond){
                    1, // 给value
                    2, // 给number
                    {most[0], most[1], most[2], most[3]...}};
  //给arrats
```

当然也可以构造完成之后再返回实体，不过这么做不如上面写的效果好，原因前方已经提过。

稍微修改一下结构体，又是另一番情况：

```
    typedef struct compond{
            int value;
            int number;
            int arrays[]; /* 这里不再显式声明大小，也就无法构造实体 *
  /
    }compond;
```

这个方式很像前方提到的 前桥和弥的 越界结构体 的例子，只不过这个是一个在C标准允许的情况下，而前桥和弥则是利用一些C语言标准的漏洞达到目的。

在使用这种结构体的时候，首先要为其动态分配好空间，之后通过指针进行操作，也增建了内存泄漏的风险，所以仁者见仁智者见智了：

```
  compond* ptocom = malloc(sizeof(compond) + num_you_want * 
sizeof(int));
  /* 这样就成功分配了足够的空间 */
  ptocom->arrays[0] = some_number;
  ...
  free(ptocom);
  ptocom = NULL;
```

这其实并不是这种机制的目的，我觉得这种复合文字机制的最大用处还是在于消除艰涩难懂的函数调用

例如有一个函数的参数列表及其之长，我们就应该考虑使用新机制结合结构体，来对这个函数重新修饰一番：

```
  int bad_function(double price, double count, int number,
                   int sales, Date sale_day, Date in_day,
                   String name, String ISBN, String market_n
ame,
                  ); /* 实现省略 */
```

这种函数，在陌生的他人拿到之后，一定头疼不已，可以对它进行一些处理，来减轻使用时候的苦恼：

```
  /* 首先使用宏进行包裹 */
  #define good_function(...) {\
  /* 使用这个宏作为接口，可传入不限个数的参数 */
```

接下来定义一个结构体，用于参数的接收。

```
/* 接收参数的结构体 */
typedef struct param{
double price;            /* 销售价格 */
double count;            /* 折扣 */
int    number;            /*总数量*/
int    sales;             /*销售数量*/
Date   sale_day;         /* 销售日期 */
Date   in_day;            /* 进货日期 */
String name;           /* 货物名称 */
String ISBN;           /* ISBN号 */
String market_name;    /* 销售市场 */
}param;
/* 并配上文档说明每个参数的作用 */
```

其次继续完成宏

```
/* 此时将函数的声明改为： */
int bad_function(param input);
/* 宏 */
#define good_function(...) {\
    bad_function((param){__VA_ARGS__});\
}
```

这就完成了包裹

使用的时候：

```
good_function(.price = 199.9, .count = 0.9,
              .number = 999, .sale = 20 /*and so on*/)
```

也可以在宏利使用默认参数，以此来减少一些不必要的工作量，达到像其他高级语言一样的函数默认参数的功能。当然如果不添加默认的值，则会按照标准将其值初始化为 `0` 或者 `NULL` .

```
#define good_function(...) {\
    bad_function((param{.price = 100.0, .count = 1.0, __VA_ARGS__})); \
    /* 假设想要设置默认价格为100， 默认折扣为 1.0 */\
}
```

较之**C89(C90)**的提取可变宏参数要来的更加灵活及**"**高效**"**。

至于 `__VA_ARGS__` 宏的较为官方的用法，前人之述备矣，就不在这里记录了。

C11之 _Generic

只看名字就能明白这是C语言支持泛型的兆头。

好像很有意思

不过某些地方依旧有些限制，比如对于选择函数方面。

```
/* -std=c11 */
void print_int(int x) {printf("%d\n", x);}
void print_double(double x) {printf("%f\n", x);}
void print(){printf("Or else, Will get here\n");}
#define CHOOSE(x) _Generic((x),\
                  int : print_int,\
                  double : print_double,\
                  default : print)(x)
```

调用它

```
int main(void)
{
    CHOOSE(11.0);  /* 11.000000 */
    CHOOSE(11.0f); /* Or else, Will get here */
    return 0;
}
```

缺点就在于，`:` 后面无法真正的调用函数，而是只能写上函数名或者函数指针，当然为了突破这一点可以使用宏嵌套来间接实现这一点，但是归根结底，无法在 `:` 后面调用函数。

```
#define CHOOSE(X) _Generic((x), \
                            int : prinf("It is Int")\
                            double : printf("It is double"))(x)
/* Compile Error! */
```

这样做会导致编译错误，编译器会告诉你 `CHOOSE` 并不是一个函数或者函数指针，看起来错误很无厘头，实际上一想，你要是在 `:` 之后调用了函数，那么左后一个括号该如何自处，唯一的办法就是返回函数指针：

```
typedef void (*void_p_double)(double);
typedef void (*void_p_int)(int);

void print_detail_double(double tmp){
    printf("The Double is %f\n", tmp);
}
void print_detail_int(int tmp){
    printf("The Int is %d\n", tmp);
}

void_p_int print_int(){
    printf("It is a Int! ");
    return print_detail_int;
}
void_p_double print_double() {
    printf("It is a Double! ");
    return print_detail_double;
}

void print_default(){printf("Nothing Matching !\n");}
#define CHOOSE(x) _Generic((x),\
                           int : print_int(x),\
                           double : print_double(x),\
                           default : print_default)(x)
```

调用：

```
CHOOSE(11);   /* It is a Int The Int is 11 */
CHOOSE(11.0);  /* It is a Double The Double is 11.000000 */
CHOOSE(11.0f); /* Nothing Matching ! */
choose(11l);  /* Nothing Matching ! */
```

> 对于宏而言，最新的编译器支持， `#program once` ，将这个放在头文件中，就代表该头文件只编译一次，也就是说，可以替代原有的老式 `#ifdef` 的三段式保护，具体编译器支持请查询各编译器。

函数返回实体

- 许多年前，在C编程的普遍常识是，返回指针，而不是一个实体。
- 但是现在，在这个**C99(C11)**世纪，早已经打破这个局限，无论是从程序员编写的语法角度看，亦或者是从编译器的优化角度看，都不在需要特地的将一个实体表示为指针进行返回。

  ```
  combine* ret_struct(combine* other){
      /* 这里的参数也是指针，因为当时并不允许直接给结构体进行赋值 */
      int value = other->filed_value;
      /* SomeThing to do */
      combine* p_local_ret_com = malloc(sizeof(combine));
      /* 一系列安全检查 */
      return p_local_ret_com;
  ```

  这在当下自然也是可以的，而且会有不错的性能，但是。但是这也是C语言最令人诟病的地方，你却深深的踏了进去。

  尽量少用 `malloc(calloc, realloc)` 之类的内存操作函数，是现代C编程的一个指标，在这个函数中，我们没有办法保证分配出去的内存能够回收(因为就这个函数而言并没有回收这个内存)，虽然现代计算机(非特殊机器)的内存已经不在乎那几十个甚至几百个中等结构体的内存泄漏，但是内存泄露依然是C语言最严重的问题，没有之一。

  我们该做的就是尽量减少风险的发生率：

```
combine ret_struct(combine other){
    /* C99之后，我们就开始允许直接给结构体赋值，
            意味着可以直接返回结构体了 */
    combine loc_ret_com; /* 如果没有复合的结构体成员的话，各成员
会自动初始化为0，不必担心初始化问题 */·
    /* Do SomeThing to 'loc_ret_com' with 'other' */
    ...
    return loc_ret_com;
}
/* main */
int main(void)
{
    combine preview = {...};
    combine action = ret_struct(preview);
    return 0;
}
```

这么做的目的自然是为了让我们的风险降到最低，让系统栈帮我们管理内存，包括创建**->**使用**->**回收，这个过程(就像被其他语言所津津乐道的**GC**机制，实际上C语言程序员可以选择自己实现一个垃圾回收机制，在本系列的最后面可能会做一个简易的回收机制供大家参考，但是首先让我们看完风景，再用一个实际程序串联起来后，再去考虑**GC**)不需要你来操心。

但是这真的是最好的形式了吗？

让我们回想一下**C**语言在调用函数的时候发生的某些事情，因为最开始的我们是从 `main` 函数的调用开始我们的程序.

- 也就是说，系统在栈上位这个函数分配了空间
- 紧接着我们调用了函数 `ret_struct`
- 调用之后，为了保存现有状态，栈里会被压入许多信息，包括当下 `main` 的位置以及 `ret_struct` 的各种参数等等，其中有一个东西就是返回地址
  - 这个被压入的元素保证了在执行完 `ret_struct` 之后我们能够顺利的返回 `main` 调用它的位置继续执行
  - 这个和我们要讲的有什么关系呢？
- 没关系我会乱说 `= =`
- 一般来说，在函数返回一个值(把所有对象，值都称为值)时，由于这个值

是在函数中创建的(无论是传入的参数，还是在函数里创建的非 `static` 对象，即便是 `static` 或者 `全局变量` 情况也是一样只是不符合这个假设结论罢了)，所以在函数结束后，栈空间被回收，它就被默认的销毁了(可以参考前桥和弥的书里有这个的解释，实际上值并没有眞正被销毁了，但是不允许再用，否则视为非法)，但是我们是怎么接收到函数的返回值的？

- 当然是因为程序帮你拷贝了一份这个值的副本的原因啊。
- 而这个副本再使用过以后就会立即被销毁，那么我们如果像上方那么返回一个结构体的话会发生什么应该就很清晰了： `复制副本->销毁本地的原身->将这个副本的值赋给外部接收的变量(没有则销毁)->销毁副本`
- 这有什么问题，难道还有更好的方法？

> 那自然有啊

- 现代科技飞速发展，编译器也不甘示弱，只要你外部有接收的地址，在(不开优化的情况下，开了优化也可能因为版本问题或者某些不可抗力而不优化)直接 `return` 对象的情况下，是可以省去副本的操作的

- 也就是说：

```
/*改写上方代码*/
combine ret_struct(combine other){
    other->filed_value = ...;
    /* SomeThing to other */
    return (combine){ .filed_value = other->filed_value
                        ...};
}
```

如果这么写，编译器就知道，哟！你是想要把这个对象放到外边使用是吧，那我懂了，就直接找到外边接收这个值得变量地址，不再创建副本(其实还是创建，只不过不再销毁而已)，而是在那个变量地址中写入这个对象。

- 这就实现了让系统帮你管理内存的目的，而不是担心是否没有释放内存带来的风险，而且还优化了性能，何乐而不为。

- 注：关于上方提到的 开了优化也可能因为版本问题或者某些不可抗力而不优化 这个说法是有道理的，因为大家的编译器版本都不一样，有的人用老版本那自然没有这个优化了，有的则是因为你编写的程序逻辑上的构造导

致编译器无法为此处产生如此的优化，这个请参考前方提到的书本深入理解计算机系统的优化章节。让然编译原理要是能看自然更清楚喽(ps:我还没看)

- 题外话：这个方法对于 `C++` 同样适用

0x0A-C线程和Glib的视角

C11之线程

这部分 **GCC** 并没有提供实现，也就是说GCC没有义务提供这个实现，我们只能用一些第三方的实现。

看不懂这一次GCC什么用意，都四年过去了。

所以现在在写跨平台多线程的程序时我一般选择使用 **Qt** 这个框架（C++）。

当然，C语言发展了这么多年了，自然少不了自己的第三方库，实际上标准库只提供了很小的一部分内容，甚至连某些常用的数据结构都未曾实现，我们该一直反复造轮子吗？

> 当然不！

在这个C的变成世界里，有许多实用的库，其中最有名的且最通用(跨多个平台的实现包括**Windows**，要知道很多实用的编程库都不提供**Windows**的实现)就是GLib这个库，其中就有实现线程的部分。

但是，因为这是中文的，看的人自然不是歪果仁，中国编程新手大都还是习惯用 **Windows** 环境，也不做强求，仁者见仁智者见智，后续会有一个程序作为例子，其中简单的应用了多线程的知识来写一个备份软件，线程的实现是用的 **Windows** 自己的接口，所有这些接口都能在 **MSDN** 里查找到相应文档。

Glib库在Windows下的配置

> 之所以不说 ***nix** 系统下的配置是因为，哪里的配置太无脑了，特别是**Ubuntu**，一句命令+有网络基本就配置完毕了。

- 使用的是稳定版的**2.28.8**版本，截至目前可用的最高稳定版本为**2.46.x**版本
- 将预处理配置好一些步骤的**glib**打包放在我的网盘中，可以直接下载，添加IDE的路径就能使用，这是对于 **Visual C++** 系列编译器能用，如果用 **MinGW** 系列的编译器就需要重新编译
- 如果想自己配置，也可以前往这个网址进行下载，或者前往**GNU**项目主页下载最新的源码以及工程文件自我编译，方式有很多，不使用现有二进制而自行选择编译的大概莫过于想使用MinGW，在MinGW项目的主页也有介绍

- 如果资源太少，可以参考如何编译**GTK**项目的方法，因为**GLib**的前身便是**GTK**的一部分，只不过后来独立出来了。
- 微软的宇宙级编译器**Visual Studio**对于**C89(C90)**之后的标准并不支持，但是对其中的特性却早早进行了实现(即没有可开启标准的选项，但是新标准所说的特性它都拥有，都能够使用，甚至还要更加超前)
- 故接下来的备份程序将使用**Visual Studio 2013** 进行编写。
- 配置**glib-2.28.8**
- 下载编译好的二进制包，预处理好(某些操作，不多说，网上有教程，记得用谷歌，或者到博客园里找类似的，但是版本比较老可能和我用的有一些出路，但可以依着葫芦画瓢)以后，将路径配置到工程里：
    1. 创建一个Win32程序，并且在属性管理器(左侧栏下部寻找)中创建属性表(Debug和Release各创建一个，设置都相同即可)
    2. 打开新建的属性表
    3. 通用属性**->VC++**目录**->**包含目录**->**编辑 添加下载下来的文件中的 `glib\glib2.28\include` 目录，不放心的还可以再添加一个 `glib\glib2.28\lib\glib-2.0\include` 目录
    4. 通用属性**->VC++**目录**->**库目录**->**编辑 添加 `glib\glib2.28\lib` 目录
    5. 通用属性**->**链接器**->**输入**->**附加依赖项 添加 `glib\glib2.28\lib` 目录下的所有 `.lib` 文件，即将这些文件的名字都手动输入进去，如果使用我的这个版本的话那就是
    6. `gio-2.0.lib glib-2.0.lib gthread-2.0.lib gmodule-2.0.lib gobject-2.0.lib`
    7. 通用属性**->C/C++->**代码生成**->**运行库开启 `多线程/MT`
    8. Okay！成了

休息一下

- 其实对于C程序员而言，最重要的莫过于使用一系列开源库，而不是对新标准的追求，因为越低的标准越容易跨平台，对于库而言这是先辈总结的一系列实用的数据结构和算法，甚至是实用的框架。我们不一定需要配置他们，而是从里面吸取一些他们的技术，转为自己的代码，毕竟库对于很多程序员编写的程序来说都大材小用了，但有时候又不得不使用一些必要的数据结构和算法。
- 在大学的这几年里，也许是因为不过是一个吊车尾的一本，所以我无法感受到老师教授带来的教导，但是也使得我深深的接触到了开源，开源给予了我很

多，比如更开阔的编程思路，更广阔的心胸，更有进步的动力，更多的小伙伴。当然也知道自己的渺小。

- 是很多人(比如知乎的回答人和提问者)，都提到要多观看**C**的源代码， 但是这对于初学者，甚至现在的我感觉也不是一件容易的事，更遑论初入门的同学了，特别是对于许多上个世纪的大神，为了节省空间以及提高效率，简直是无所不用其极！虽然某些用法能够被现代接受，但是你能在第一眼就看出来，为了构造一个红黑树节点，把树的指针和节点的颜色信息都隐藏在一个指针地址里吗？

```
/* 假设有一个节点的指针 p_node */
node_color = p_node->node_color & 1; /* 原理就是用最后一位bit来存储颜色 */
```

其中在 Linux 里, `p_node->node_color` 被设定为无符号的长整形，以整数型式存储指针和颜色信息，而不是用指针类型。

```
node_pointer = (node_type*)p_node->node_color & ~3;/* 清除最后两位上的bit的值 */
```

也就是清除颜色信息，留下的就是指针的值，即地址。

为什么呢，只要我么能够保证节点的创建位置是**32**位**/64**位对齐的，我们就能够保证它的最后两位/三位是空的，绝对不会被使用的。

```
/* 32位 */
sizeof(void*); /* 是 4 */
/* xxxx xxxx xxxx xxxx xxxx xxxx xxxx xx00 */
/* 64位 */
sizeof(void*); /* 是 8 */
/* 前方省略48位 xxxx xxxx xxxx x000 */
```

意思就是，对于指针而言，因为编译器要保证寻址的高效所以它在给分配地址的时候，会对齐内存中的地址，按照指针大小的倍数对齐，这就会导致不同位的程序的指针变量的值中有几个 `bit` 会没有使用，则用它来存储。

- 具体的情况，网路上的详细解说十分之多，开一个头就好。但是这真的是我们一开始就应该接触的吗？

> 是

- 怎么说，在很多的时候，C语言给我们的函数都不够安全可靠，但是在我们无法使用新标准提供的函数的情况下(十分常见)我们该如何做呢？当然是自己写，怎么写更完美，自然是看看别人怎么写，而不是自己一抹黑的乱来，因为事实证明，自认为好的到最后都会摔一跤，虽然不是坏事。
- 最简单的做法便是用宏包裹一下，做一些预处理，或者对于宏机制不太喜欢的人会选择用一个函数进行包裹，也未尝不可。

注

- 写在最末尾，填几个前面挖的坑。
- 不知道是不是故意的，一般**GNU**项目的子主页面上，找不到(很难找到)对应的项目的下载地址，也就是光看着介绍如何如何牛，如何如何好用，但就是不告诉你去哪里下，这时候，首先确认你要下的这个软件的名字，然后去**GNU**项目首页里的程序列表里找，在哪里一定能找到，而不是在那些介绍页面乱点，结果根本找不到。
- 最典型的就是一个叫做**GMP**的开源软件，用来自行编译**MinGW**用的依赖，希望能警醒各位。
- 之所以用**2.28.8**而不是**2.46.x**是因为我实在不想自己在**Windows**上编译了，因为大部分时候，写程序都是在 **Linux** 上，所以就偷懒一下。
- 对于我的文件是不是有毒，我说有毒，有一种叫做叫你再用**Windows**编程的毒。
- 好吧其实我承认**Visual Studio**的确是宇宙无敌的编译器。

末尾

- 接下来的第三部分我会用一个备份程序来贯穿
    - 操作系统：**Windows**
    - 跨平台：否
    - API调用： `Win32 API`
    - 编译器： `Visual Studio 2013`
    - 语言： `Pure C Programing Language`
- 会在里面介绍一下，常在开源代码中看见的一些奇怪的东西，例如

```
#ifdef __cplusplus
extern "c" {
#endif
...
#ifdef __cplusplus
}
#endif
```

这到底是什么

第三部分

> 实用C编程以及程序实战

0x0B-C语言错误处理

- 三个必要的头文件

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
```

- 一个声明

```
extern int errno;
```

- 四个重要函数

```
int ferror(FILE* stream);
int feof(FILE* stream);
char *strerror(int errnum);
void perror(const char *s);
```

- 以上是在错误处理中常用的操作，前两者是必要包含的，最后的函数则是酌情使用

1. `ferror` ：
    - 检查流中是否有错误，如果有则返回一个非 `0` 值，如果没有错误则返回 `0`
2. `feof` ：
    - 用于检查是否到了文件尾，经常用于文件的读取工作，读取到文件尾则返回非零值

```
input = fopen("file.in", "r");
while(!feof(input))
{
    fscanf(input, "%s", str);
    ...
等同于
do{
    rtn_value = fscanf(input, "%s", str);
    ...
}while(rtn_value != EOF);
```

3. `strerror` ：

   - 当某个函数或者操作触发了设置 `errno` 变量的时候，我们可以立即使用该函数进行显示输出

   ```
   strerror(errno);
   输出： Too many open files
   ```

4. `perror` ：

   - 与上方的函数相同，也是用来输出错误的，只不过它自动使用 `errno` 变量，并且在自前方添加所传入的参数以及一个冒号

   ```
   perror("Now");
   输出： Now: Too many open files
   ```

- 同样的，对于自己的错误处理，可以设计一个小体系来满足自我需求
  - 通过返回值

  ```
  /* fun.h */
  void   fun_error(size_t errnum);
  size_t test_fun(int arg);
  ```

  在此我们可以让外界接触不到真实的包含有错误的数组或其他数据结构，而是将其放于 `.c` 文件中隐藏起来，通过一个函数来进行访问。

- 隐藏错误实现

  ```
  /* fun.c */
  static const char * fun_err[] = { "Computing nagative!"
  ,
                                    "Invalid argument"
  ,
                                    "Bad result" };
  size_t test_fun(int arg) { ... }
  void   fun_error(size_t errnum) { ... }
  ```

  对于需要进一步规格化的设计来说，可以使用 `enum` 或者 `#define` 来设定每个错误返回值的名字

- 设定名字

  ```
  /*fun.h*/
  #define ERR_CMPUTING 0
  #define ERR_INVARG   1
  #define ERR_BADRLT   2
  /* 或者 */
  enum { ERR_COMPUTING = 0,
          ERR_INVALIDARG,
          ERR_BADRESULT };
  ```

  如此我们便可以在函数中使用错误的名字代号，而不是去记下每个数字的含义。对此也可以不使用数组这个数据结构来保存错误信息，而采用 `switch` 在代码中实现错误的处理

- `switch` 错误选择·

```
/*fun.c*/
/*
static const char * fun_err[] = { "Computing nagative!"
,
                                     "Invalid argument"
,
                                     "Bad result" };
*/
...
void fun_error(size_t errnum)
{
    switch(errnum)
    {
        case ERR_CMPUTING /*ERR_COMPUTING*/:
            fprintf(stderr, "...");
            break;
        case ERR_INVARG /*ERR_INVALIDARG*/:
            fprintf(stderr, "...");
            break;
        ...
        default :
        ...
    }
    return;
}
```

如此做的好处便是，不再需要去安排数组的个数，而是直接在代码中展示。并且可以加上一些预处理，来实现开关错误显示。

- 开关错误

```
/* fun.c */
...
void fun_error(size_t errnum)
{
#if !defined(NOT_DEBUG)
    switch(errnum)
    {
        ...
    }
#endif
    return;
}
```

- 慎重使用
  - 在我设计程序的过程中，尽量让错误提示减少，或者说用户不应该接触到错误以及处理错误，所以当一个程序没有必要提供给用户错误信息的时候，才是眞正的完整程序，至于程序失败那又是量一种境况，例如备份程序可能由于文件的属性，而复制失败，这是不可避免的，此时将失败信息写入文档，呈现给用户即可。

0x0C-开始行动

写在最前方

- 对于线程的概念以及意义，我说一些
    - 现在我只说，一个多线程的技术对于C语言程序而言就像，原本只有你一个人在干活，现在突然有许多人愿意追随你，变成了以你为核心的多人合作模式，共同完成同一个任务。
    - 导致的结果就是，这个任务被切分成多个模块，有可能很多人一起做同一个模块，有可能某些模块只有一个人在做，这就带来了什么问题呢？
    - "我" 对这个任务的掌控力度变低了，试想本来就是我一个人在开发这个程序，程序的每部分都是我一个字符一个字符敲上去的，自然了解程序的每个部分。但是现在，突然多了一些人来和你一起完成这个任务，必然导致有一部分程序代码不是由你亲自写下去的，虽然你指示他们怎么做，做出什么样的功能。
    - 但是毕竟不是你亲自写的，所以他们在协调工作的时候，很大可能性会出现问题，这个问题在多线程变成里面就是资源争夺问题，也就是同一个内存块， 在同一时刻被多于一个的线程访问，那么该如何处理呢？
    - C语言到现在已经支持多线程(然而零零散散的，虽然C11标准支持多线程库)了，并没有办法实际用在编程里，我一般使用 各平台的 **API** 或者 **Glib** 进行多线程编程，前者是不需要配置环境直接上手，缺点就是无法跨平台(万恶的**Windows**)。后者则是需要配置环境，且这个库说实话的确很臃肿，对于我们即将写的这个备份程序有些杀鸡用牛刀。
- 好吧，如果硬要说的官方一点那就是：

    - 应用程序，进程，线程的区别：
    - 应用程序是一组数据和指令。
    - 多进程就相当于启动了多个应用程序，彼此之间独立(虚拟内存)
    - 进程就是应用程序运行起来的名称，两者的区别在于进程是有状态的，即它会变。
    - 线程和进程十分相似，都有状态，但是它的状态比进程少，并且线程之间可以十分轻易的共享数据，而这点是多进程无法比拟的(进程间共享数据开销极大)。
    - 状态指的就是程序运行起来产生的一些值，因为是运行后产生的所以形象的叫他们状态，例如寄存器里的值，栈**(**而不是堆，线程并没有自己的堆**)**的值

- 看完上面的解释，就准备开始着手写程序了

写在中间

- 首先我使用的是 **Visual Studio 2013**作为编译器
- 创建 **Win32**控制台应用程序
- 记得创建**C**源文件
- 如果不小心点成**C++**也没事，改成**.c**就行。
- 创建一个入口源文件 `Entery.c` ，用于放置 `main` 函数

```
    #include <stdio.h>
    #include <stdlib.h>

    int main(int argc, char* argv[])
    {
        system("pause"); /* 因为Windows的命令行命令对大小写不敏感，所以命令无所谓大小写,从环境变量的配置就能知道这一点 */
        return 0;
    }
 因为等一下需要构造一个字符显示界面模拟GUI的功能，故添加`#include <stdlib.h>`

 当然并不是只因为这个原因。但是现在是为了使用其中的`system()`函数调用系统命令。
```

- 好，那么接下来...
- 等一下，要直接写程序了吗？程序功能呢？程序结构呢？从哪里开始写呢？
    - 这些都没有考虑啊！！
- 所以，停下来我们好好构思构思！
    - 问题：
        1. 我们要做什么？ ： 一个多线程的备份程序
        2. 我们要怎么做？ ： 这个问题有点大，应该拆开
        3. 我们从哪里开始做？ ： 这个问题比较好回答。
    - 回答：
        1. 略

    2. 我们要实现把某个路径备份到某个特定路径
        - 首先要实现路径的设置，也就是说由键盘输入路径
        - 得到了路径之后，要得到该路径下所有文件夹和文件的信息(在*nix下这俩玩意儿都是一个东西)
        - 行了就这么简单
    3. 先从输出所有文件和文件夹的结构开始
  - 问题：
    1. 怎么得到文件和文件夹的信息？
  - 回答：
    1. MSDN放在网上，虽然现在不提供离线版本了，但是也没有被墙啊。所以不懂的时候就一个字，查**API**文档！对于十分基本的**API**，MSDN甚至会给出示例代码。恰好，这就是一个。
- 题外话
  - 其实会写这个程序没什么大不了，重要的是学习能力，一个自我学习的能力，就比如当我不知道一个东西，也无从下手的情况下，我应该进行联想，联想可能与他相关的各种方面，并且亲自去查，不厌其烦的查，遍地撒网，重点捞鱼。
- 首先我们需要实现一个功能，就是遍历输出路径下的所有文件夹和文件
- 在这之前，我们先设计一下界面，稍后... `3, 2, 1` ，出现！

```
while(1)
{
    system("cls"); /* 系统的清屏命令 */
    do{
        puts("-----------------------------------------------------");
        puts("-----------------------------------------------------");
        puts("That is a System Back Up Software for Windows! ");
        puts("List of the software function : ");
        puts("1. Back Up ");
        puts("2. Set Back Up TO-PATH ");
        puts("3. Read Me ");
        puts("4. Exit ");
```

```
            puts("-------------------------------------------------");

            puts("Your Select: ");
            fscanf(stdin, "%d", &select);
            getchar(); /* 读取上方 fscanf 留在流里面的换行符 '\n' */
            }while ((select < 1) || (select > 4)); /* 如果选择无效 */
            system("cls");

            switch(select)
            {
            case 1 :
                break;
            case 2 :
                break;
            case 3 :
                break;
            case 4 :
                exit(0); /* 退出程序 */
            default :
                break;
            }/** switch(select) **/
        }/** while(1) **/
        system("pause");
        return 0;
```

突然出现的这一段代码就是设计的界面，其实很简单，看看就懂了，不再多说。

英文莫怪。

- 紧接着，我们来实现第一个功能，显示结构，让我们吧这个功能函数叫做 `show_structure`
    - 新建头文件 `showFiles.h`

```
/*头文件包裹一定要切记*/
#ifndef INCLUDE_SHOWFILES_H
#define INCLUDE_SHOWFILES_H
/* 代码写在里面，这样就不会发生重定义，也能节省资源 */
#endif
```

- 新建源文件 `showFiles.c`

```
- 记得包含头文件`#include <showFiles.h>`
```

- `showFiles.h` 稍后进行解释

```
#include <stdio.h>
#include <stdlib.h>
#include <Windows.h> /* Windows API */
#include <string.h>  /* 字符串函数 */
#include <tchar.h>     /* _ftprintf */
#include <setjmp.h>  /* setjmp, longjmp */

#define TRY_TIMES 3  /* 重新尝试获取的最大次数 */
#define MIN_PATH_NAME _MAX_PATH /* 最小的限制 */
#define LARGEST_PATH_NAME 32767 /* 路径的最大限制 */

/* 我们需要在这里面包含函数的声明 */
/** 加上文档注释，不太喜欢死板的硬套，选择自己觉得重要的信息记录吧
 * @version 1.0 2015/09/28
 * @author  wushengxin
 * @param   from_dir_name 源目录，即用于扫描的目录
            depth         递归的深度，用于在屏幕上格式化输出，每次增加 4
 * @function 用于输出显示目录结构的，可以改变输出流
 */
void show_structure(const char * from_dir_name, int depth);
```

题外话，在**Visual Studio**中，会强制要求你使用他们编写的安全函数,例如 `fscanf_s` ，如果你不想用的话，那就将它关闭吧，具体怎么操作，就当是一个小问题留给你自己。

- `showFiles.c` 先不写太多，这里比较重要的是写法

```
  /* 首先是需要的一系列变量 */
  int times = 0; /* 用来配合 setjmp和longjmp重新获取句柄HAND
LE的 */
  /** 操作时获取文件夹，文件信息的的必要变量 **/
  HANDLE file_handle;
  WIN32_FIND_DATAA file_data;
  LARGE_INTEGER file_size;
```

  `file_handle` ：文件的句柄，后期操作的主要对象

  `file_data` ：文件的信息，各种属性

  `file_size` ：文件的大小有可能非常大，需要使用特定的结构体保存

- 到这里我们停下来，因为下一步我们要去实现获取路径的操作。
  - 问题：
    1. 我们要怎么样获取路径
    2. 我们获取到的路径要怎么存储
    3. 存储的路径要符合什么格式
  - 回答：
    1. 有两个路径：备份的来源路径，备份的目的路径。前者用键盘输入，后者在程序内部首先指定一个。
    2. 这里有两种方案：用系统栈存，用堆存
       - 前者是方便的内存管理，完全不用程序员操心，但是栈的大小比较小，一般只有几兆而已，这也是为什么递归容易爆栈的原因。速度较快
       - 后者是近似巨大的内存上限，对于32位系统的**Windows**应用程序而言，可以有 `2~3GB` 的分配空间(**4GT**机制)，64位就更为可怕了(**Windows8** 最大有 `512GB` , **Windows7** 最大有 `192GB` ，服务器系列大概是 `1~2TB`)。速度较慢
       - 不熟练的程序员应该尽可能选择前者。

        - 这里采用后者，前者的代码也会一并附上。
    3. 对于微软API处理自己的**Windows**路径，一般要求末尾以 `/` 或者 `\` 结尾，前者在C语言中不是转义，所以比较好存储，如果需要使用后者，可以选择如此 `\\` 就行了。

    ```
    char dir_path_1[PATH_MAX] = "../";
    char dir_path_2[PATH_MAX] = "..\\";
    /* 两种效果一致，且占的空间也相同 */
    ```

    - 注意
        - 这里有涉及到一个问题，那就是**Windows**下的路径限制，在 `windef.h` 中定义了一个常量 `PATH_MAX` 的值为260，也就是说最大的路径长路为 `260字节` ，但是如果我们的路径名超过了这个长度怎么办呢？
        - 这里直接给出了解决办法，就是添加前缀 `\\?\` ,这样长度限制就增加到了 `32767` 了
        - 此处不予以实现，日后遇见情况的话可以当作一个解决的办法。
- 解决了上一个关于路径的问题，我们就需要考虑一下如何设计实现这个功能，首先要达到模块化的目的，即尽量减少每个函数的功能。

    - 问题
        - 都需要什么功能？
    - 回答
        - 一个主要的函数用来递归(也可以用循环，循环的好处就也在于不容易爆栈)
        - 一个用来专门给路径分配空间的函数
        - 一个用来释放分配空间的函数
        - 一个对输入的路径进行处理的函数，让路径变得规范
- `showFiles.h` 变个魔术 `3, 2, 1` ，添加了如下代码

```
    /**
     * @version 1.0 2015/09/28
     * @author  wushengxin
     * @param   src 外部传进来的，用于向所分配空间中填充的路径
     * @function 用于在堆上为存储的路径分配空间。
     */
    char * make_path(const char * src);

    /*
     * @version 1.0 2015/09/28
     * @author  wushengxin
     * @param   src 外部传进来的，是由 make_path 所分配的空间，
将其释放
     * @function 用于释放 make_path 分配的内存空间
     */
    void rele_path(char * src);

    /*
     * @version 1.0 2015/09/28
     * @author  wushengxin
     * @param   src_path 用于 FindFirstFile()
                src_file 用于添加找到的目录名，形成新的目录路径
     * @function 用于调整用户从键盘输入的路径字符串，使他们变得一致
，便于处理
    */
    void adjust_path(char * __restrict src_path, char * __
restrict src_file);

    /*
     * @version 1.0 2015/09/28
     * @author  wushengxin
     * @param   src 外部传入的，用于调整
     * @function 用于替换路径中的 / 为 \ 的
     */
    void repl_str(char * src);
```

具体功能在文档里已经写的很清楚了，唯一要解释的就是最后两个函数，本来是一体的，后来被我拆开成了两个函数，为了也是功能更加清晰

倒数第二个函数 `adjust_path` 的作用是将路径处理成符合 **Windows** 函数 `FindFirstFile` 的要求.aspx)可以具体看看。

- `showFiles.c` 继续 `show_structure` 的实现
    - `shows_tructure`

    ```
          size_t length = strlen(from_dir_name);
          char * dir_buf = make_path(from_dir_name);  //路径
          char * dir_file_buf = make_path(from_dir_name); //
    文件
          if (dir_buf == NULL || dir_file_buf == NULL)
              return; /* 如果分配失败就结束函数 */
          adjust_path(dir_buf, dir_file_buf); /* 调整路径和文件
    格式到标准格式 */
          repl_str(dir_buf);
          repl_str(dir_file_buf);
    ```

    这是调用 `WINDOWS API` 之前的所有操作，来一一实现他们

    首先是分配空间给路径

    - `make_path` : `24 lines` .

    对于这个函数的功能便是，为需要存储的路径分配空间。

```
        int times = 0;
        size_t len_of_src = strlen(src); /* 需要分配的长
度 */
        size_t len_of_dst = MIN_PATH_NAME;

        if (len_of_src > MIN_PATH_NAME - 10) /* \\?\ //
* 8个字符 */
        { /* 这里用了10这个神奇的垃圾数，所以必须做一点注释，
以防忘记 */
            len_of_dst = LARGEST_PATH_NAME;
            if (len_of_src > LARGEST_PATH_NAME - 10)
            {
                fprintf(stderr, "The Path Name is large
r than 32767, Which is not Support!\n%s", src);
                return NULL;
            }
        }
        setjmp(alloc_jmp); /* alloc_jmp to here */
        char * loc_buf = malloc(len_of_dst + 1);
        if (loc_buf == NULL)
        {
            fprintf(stderr, "ERROR OCCUR When malloc th
e memory at %s\n Try the %d th times", __LINE__, times+1)
;
            if (times++ < TRY_TIMES)
                longjmp(alloc_jmp, 0); /* alloc_jmp fro
m here */
            return NULL;
        }
        //sprintf(loc_buf, "\\\\?\\%s", src); /* 作为日
后的扩展 */
        strcpy(loc_buf, src);
        return loc_buf;
```

对于 `10` 这个数的考虑是，至少留出 `8` 个空位给所说的字符，加 `2` 凑整。

对于函数 `malloc` ，在这里没有进行包裹，是因为这只是一个预热的功能，后期在实现备份的时候，会对它进行包装，也使得错误处理的代码隐藏，让函数功能更加清晰。

- `adjust_path` : `16 lines`

  其次是调整路径的函数，功能就是调整路径

```
  size_t length = strlen(src_path); /* 两个参数的长度在此函数调用之前必定一致 */
  if (length == 1) /* 处理情况为，当用户输入的是根目录的情况 例如: C */
  {
      strcat(src_file, ":/");
  }
  else if (src_path[length - 1] != '\\' && src_path[length - 1] != '/')
  {
      strcat(src_file, "/");
  }
  else
  {
      src_path[length - 1] = '/';
  }
  strcpy(src_path, src_file);
  strcat(src_path, "/*");
  return;
```

  当用户输入的是一个字符的根目录，我们要将其处理为 `C:/` 这样的形式

  当用户输入的是不带 `/` 结尾的，我们需要将其添加上 `/`

  当用户输入以 `\` 结尾的路径时，将其替换为 `/`,虽然后方又全部换成了 `\`

  将目录处理为带 `/*` 结尾的，以达到 **API** 的要求

  `src_file` 用于将目录下的子目录名连接。生成新的目录。 `src_path` 用于递交给 **API** 扫描目录下的所有文件和文件夹。

- `repl_str` : `7 lines`

```
    size_t length = strlen(src);
    for (size_t i = 0; i <= length; ++i)
    {
        if (src[i] == '/')
            src[i] = '\\';
    }
    return;
```

不再赘述这个函数的功能

到此处，所有在第一个 **Windows API** 之前调用的函数都实现了，接下来要做什么？

- 当然是调用**API**函数啦
  - `show_structure`

```
  /* 开始调用 Windows API 获取路径下的文件和文件夹信息 */
  setjmp(get_hd_jmp);
  fileHandle = FindFirstFileA(dir_buf, &fileData);
  if (fileHandle == INVALID_HANDLE_VALUE) /* 如果无法获取句柄超过上限次数，就退出 */
  {
      fprintf(stderr, "The Handle getting Failure! \n");
      if (times++ < TRY_TIMES)
          longjmp(get_hd_jmp, 0);
      return;
  }
```

对于这一段代码的解释，其实核心就是第二句代码，其中的函数 `FindFirstFileA` 需要解释一下。

在 **Windows API** 文档 **MSDN** 中介绍的是 `FindFirstFile` ，但是某些情况下(定义了UNICODE宏，不知道有没有记错)，这个官方提供的接口会被定义(`#define`)成 `FindFirstFileW` ，如果使用 `char *` 的 **ANSI** 字符串当成参数的话是会获取句柄失败的！并且另一个参数使用的 `file_data` 类型也是 **ANSI** 的 `WIN32_FIND_DATAA`

所以这里显式地选择调用 `FindFirstFileA` 而不是让 **Windows** 帮我们选择。

接下来我们要做的事情就是，遍历这个目录下的所有文件和文件夹，提取出来他们的信息：

```
  do{
      char * tmp_dir_file_buf = make_path(dir_file_buf);
      if (fileData.dwFileAttributes & FILE_ATTRIBUTE_DIRE
CTORY)
      { /* 如果是文件夹 */
          fprintf(stderr, "%*s%s\t<DIR>\t\n", depth, "",
fileData.cFileName);
          if (strcmp(fileData.cFileName, ".") == 0 ||  /*
 . 和 .. 便是当前文件夹和上一级文件夹, 世界上所有系统都一样 */
              strcmp(fileData.cFileName, "..") == 0)
              continue;
          strcat(tmp_dir_file_buf, fileData.cFileName); /
* 将文件名连接到当前文件夹路径之后，形成文件路径 */
          show_structure(tmp_dir_file_buf, depth + 4);
      }
      else
      {
          fileSize.LowPart = fileData.nFileSizeLow;   /*
输出大小 */
          fileSize.HighPart = fileData.nFileSizeHigh;
          fprintf(stderr, "%*s%s    \t%ld bytes\t\n", dep
th, "", fileData.cFileName,
                                               fileSize.QuadPart)
;
      }
      rele_path(tmp_dir_file_buf); /* 这个的实现稍后放出 */
  } while (FindNextFileA(fileHandle, &fileData));
```

代码是仿照 **MSDN** 提供的 官方例子.aspx)改写的。

其中第五句代码，用到了前方提到过的格式化输出的一个不常用的技巧，占位，忘记的可以回去看看在第一章

采用的方法是递归，循环的方法留给看者自己实现，思路很简单，用一个队列或者栈存放所有找到的目录和文件，依次取出直到栈或者队列为空。

以及最后一段的代码，用于收尾：

```
    FindClose(fileHandle);
    rele_path(dir_buf);
    rele_path(dir_file_buf);
    return;
```

- `rele_path` : `4 lines`

```
free(src);
src = NULL;
return;
```

- 最后在 `Entry.c` 的 `main` 函数中，在 `switch` 的 `case 1` 标签范围内，加上一些获取和处理输入的函数：(因为这里只会使用一次，故采用的是系统栈而不是在堆上分配)

```
char tmp[_MAX_PATH];
...
case 1 :
    scanf("%s", tmp);
    printf("Enter : %s\n", tmp);
    getchar();  /* 前方提到过，作用是清理标准输入流 */
    show_structure(tmp, 0);
    system("pause");
    break;
```

现在编译运行

- 成功了！对自己的代码很有信心，嗯！
- 中文路径也是可以的，别怕
- 对于超过 `260` 字符的路径没有测试，大概能猜到是不行的。但是解决方案上方也有提到。
- 这就是预热的函数，比较详细，后方代码就不会如此赘述，而是更加简洁干练

的上代码和解释

写在最后面

- 总结一个词，代码横陈，但是逻辑还算清晰
- 只是一个预热的作用，正片代码只是为了提前让思路更加清晰，且能测试出**Windows API**的某些潜在缺陷以及要求。
- 下面将开始真正的进入功能的编写。

0x0D-单线程备份(上)

写在最前方

- 源路径：即 **From-Path**，你准备要备份的资料
- 目的路径： 即 **To-Path**，你准备要存储备份的资料的地方
- 稍微回想一下，上一次写的代码，本次的任务是遍历目录及其子目录，那么这回要干的就是将上次遍历过的数据，挪一下窝，到我们想要他们去的位置。
- 这涉及到两个操作，遍历 和 拷贝，前一个动作我们在上一回已经实现了，只需做小小的改动，就能够使用。后一个动作也是需要靠 **Windows API**来完成，至于哪些，稍后再提。
- 现在先让我们完成一个魔法， `3, 2, 1!` ：

```
  do{
       puts("-----------------------------------------------
--");
      fprintf(stdout, "The Default Path is : %s \n", DEFAULT
_TO_PATH);
      fprintf(stdout, "Now The Path is     : %s \n", get_bac
kup_topath());
       puts("-----------------------------------------------
--");
       puts("That is a System Back Up Software for Windows!
");
       puts("List of the software function : ");
       puts("1. Back Up ");
       puts("2. Set Back Up TO-PATH ");
       puts("3. Show TO-PATH History");
       puts("4. Read Me ");
       puts("5. Exit ");
       puts("-----------------------------------------------
--");
```

对界面稍微有了一些改动。

新增了第三行和第四行的 系统默认目的路径和当前使用的目的路径。

新增了倒数第四行的查看目的路径历史纪录的功能。

在 `main` 函数外头需要 `extern DEFAULT_TO_PATH;` 因为引用了 `setPath.c` 里的一个全局变量。

写在中间

- 前一次我们曾经提到要让函数的功能更加清晰，为了达到这个目的，应该把可能用到的一些原生库函数包裹一下，让可能发生的错误尽量掌握在我们自己的手里
- 安全函数
    - 新建 `safeFunc.h` `safeFunc.c`
    - 考虑一下我们需要包裹的函数： `malloc` ， `free` ， `fopen` 三个库函数。
        - 为了不让后方的多线程实现产生更多的以后，不单独使用全局错误输出。
        - 让我来将他们实现一下
        - 我不会省略一些看似不必要的东西，例如注释，而是完整的呈现出来，如果觉得篇幅过长，可以选择跳跃的阅读。
        - 魔法来了，`3, 2, 1!`

        ```
        #include <stdio.h> /* size_t */
        #include <stdlib.h>
        #include <setjmp.h>
        #define TRY_TIMES 3

        typedef struct _input_para{
            char * file; /* 待打开或创建的文件名 */
            char * mode; /* 打开的模式 */
        }params;

        jmp_buf malc_jmp; /*Malloc_s*/
        jmp_buf fopn_jmp; /*Fopen*/

         /**
         * @version 1.0 2015/10/01
         * @author  wushengixin
         * @param   ... 参看结构体说明
        ```

```
              可传入任意的个数的，形式为 .file = "xxx", .
mode = "x" 的参数
   * function 用于使用默认参数，并调用函数 Fopen 进行打开操
作
   */
  #define Fopen_s(...) Fopen((params){.file = NULL, .
mode = "r", __VA_ARGS__})
  FILE* Fopen(const params file_open);

  /**
   * @version 1.0 2015/10/01
   * @author  wushengxin
   * param    sizes 输入需要分配的大小
   * function 用于隐藏一些对错误的处理，并调用malloc库函数
分配空间
   */
  void * Malloc_s(size_t sizes);

  /**
   * @version 1.0 2015/10/01
   * @author  wushengxin
   * @param   input 外部传入的等待释放的指针
   * function 用于隐藏一些对错误的处理，并调用free库函数进行
释放指针
   */
  void Free_s(void * input);
```

里面用到了一些新的特性，如果使用 `GCC/Clang` 作为编译器的，记得要开启 `-std=c11` 支持。

这几个函数就不再详细解释，而是简略说几个，接下来放上实现代码：

```
  FILE* Fopen(const params file_open)
  {
      int times = 0;
      FILE* ret_p = NULL;
      if (file_open.file == NULL)
      {
```

```
            fputs("The File Name is EMPTY! Comfirm it a
nd Try Again", stderr);
            return ret_p;
        }
        setjmp(fopn_jmp); /* fopn_jmp To there */
        ret_p = fopen(file_open.file, file_open.mode);
        if (ret_p == NULL)
        {
            if (times++ < TRY_TIMES)
            longjmp(fopn_jmp, 0); /* fopn_jmp From here
 */
            fprintf(stderr, "The File : %s Open with Mo
de (%s) Fail!\n", file_open.file, file_open.mode);
        }
        return ret_p;
    }

    void * Malloc_s(size_t sizes)
    {
        int times = 0;
        void * ret_p = NULL;
        if (sizes == 0)
            return NULL;
        setjmp(malc_jmp); /* malc_jmp To There */
        ret_p = malloc(sizes);
        if (ret_p == NULL)
        {
            if (times++ < TRY_TIMES) /* malc_jmp From H
ere */
                longjmp(malc_jmp, 0);
            fputs("Allocate Memory Fail!", stderr);
        }
        return ret_p;
    }

    void Free_s(void * input)
    {
        if (input == NULL)
        {
    #if !defined(NOT_DEBUG_AT_ALL)
```

```
            fputs("Sent A NULL pointer to the Free_s Fu
nction!", stderr);
  #endif
            return;
        }
        free(input);
        input = NULL;
    }
```

第一个函数是用外部定义的宏 `Fopen_s` 启动它，这里没有实现隐藏它。

最后一个函数中使用了预处理的机制，如果在头文件中定义了 `#define NOT_DEBUG_AT_ALL` ，这个输出将不在出现

- 安全函数已经撰写完成，接下来就是干正事了
    - `setPath.h`
        - 我们首先要将程序里保存上默认的目的路径，首先想到用常量 `#define ...`
        - 其次应该要确保当前目的路径不被其他非法的渠道访问，那就应该用一个 `static` 字符数组存储。
        - 接下来就是要提供一个函数当作接口(这里用了接口这个术语不知道合不合适)，来获取当前实际在使用的目的路径 `get_backup_topath` 。
        - 这里还需要将之前实现过的 `repl_str` ，再次实现一次，因为之前的显示功能只是测试，并不会实际应用到程序当中。
        - 完成这两个功能函数以后，再去考虑实现怎么样设置路径，存储路径，以及使用文件流操作来缓存历史目的路径

```
#include "safeFunc.h"

#define SELF_LOAD_DEFAULT_PATH "C:/"
#define MIN_PATH_NAME _MAX_PATH /* 最小的限制 */
#define LARGEST_PATH_NAME 32767 /* 路径的最大限制 */

/*
 * @version  1.0 2015/10/02
 * @author   wushengxin
 * @function 用于返回当前使用的目的路径
 */
const char * get_backup_topath();

/**
* @version 1.0 2015/09/28
* @author  wushengxin
* @param   src 外部传入的，用于调整
* @function 用于替换路径中的 / 为 \ 的
*/
void repl_str(char * src);
```

对应的实现中，会定义一个静态的字符数组，且在头文件中能够看见，很多是在 `showFiles` 里定义过的。

定义过的函数，例如 `repl_str` 需要把 `showFiles.c` 中的实现，使用 `#if 0 ... #endif` 进行注释掉，不然会发生重定义的错误。

- `setPath.c`

```
#include "setPath.h"

static char to_path_buf[LARGEST_PATH_NAME] = SELF_LOAD_DEFAULT_PATH;
const char * DEFAULT_TO_PATH = SELF_LOAD_DEFAULT_PATH;
const int LARGEST_PATH = LARGEST_PATH_NAME;

const char * get_backup_topath()
{
    return to_path_buf;
}

void repl_str(char * src)
{
    size_t length = strlen(src);
    for (size_t i = 0; i <= length; ++i)
    {
        if (src[i] == '/')
            src[i] = '\\';
    }
    return;
}
```

- 有了上面的代码，主界面就再次能够无误运行了，那么剩下的就是实现，设置目的路径，存储目的路径到本地，显示目的路径，分别对应主界面的 `2, 3` 。
- 怎么实现比较好，再开始之前，分析一下会遇到的情况：
  - 我们在得到目的路径之后，会将其拷贝给默认路径 `to_path_buf` ，并且将其存储到本地缓存文件中，以便下次程序开始时可以直接使用上一次的路径
  - 还可以使用另一个文件存储所有用过的历史路径，包含时间信息。
- 那么这就要求我们首先实现存储目的路径的功能，其次再实现设置目的路径的功能，最后实现显示目的路径的功能
- 注：两个看似无用的全局变量(`const`)是为了其他文件的可见性而设立的，且相对于 `#define` 能够省一些无足轻重的空间。

- 存储目的路径 `store_hist_path`
  - `setPath.h`

```
#include <time.h>
/**
 * @version  1.0 2015/10/02
 * @version  wushengxin
 * @param    path 需要存储的路径
 * @function 用于存储路径到本地文件 "show_hist" 和 "use_hist"
 */
void store_hist_path(const char * path);
```

  - `setPath.c`

```
void store_hist_path(const char * path)
{
    time_t ctimes;
    time(&ctimes); /* 获取时间 */
    FILE* input_use = Fopen_s(.file = "LastPath.conf", .mode = "w"); /* 每次写入覆盖 */
    FILE* input_show = Fopen_s(.file = "PathHistory.txt", .mode = "a");
    if (!input_show || !input_use)
    {
#if !defined(NOT_DEBUG_AT_ALL)
        fputs("Open/Create the File Fail!", stderr);
#endif
        return;
    }
    fprintf(input_use, "%s\n", path); /* 写入 */
    fprintf(input_show, "%s %s", path, ctime(&ctimes));
    fclose(input_show);
    fclose(input_use);
    return;
}
```

`time` 和 `ctime` 函数的使用网路上的介绍更加全面，这里不做解释。

完成了存储的函数之后，便是实现从键盘读取并且设置默认路径

- 设置目的路径 `set_enter_path`
  - 在此处需要停下来在此思考一下，如果用户输入了错误的路径(无效路径或者恶意路径)，也应该被读取吗？所以应该增加一个检查，用于确认路径的有效性。
  - `setPath.h`

```
#include <string.h>
#include <io.h> /* _access */
enum {NOT_EXIST = 0, EXIST = 1};
/**
 * @version  1.0 2015/10/02
 * @author   wushengxin
 * @function 用于读取从键盘输入的路径并将之设置为默认路径，并存储。
 */
void set_enter_path();

/**
 * @version  1.0 2015/10/02
 * @author   wushengxin
 * @param    path 用于检查的路径
 * @function 用于检查用户输入的路径是否是有效的
 */
int is_valid_path(const char * path);
```

  - `setPath.c`

```
int is_valid_path(const char * path)
{/* _access 后方有解释 */
    if (_access(path, 0) == 0) /* 是否存在 */
        return EXIST;
    else
        return NOT_EXIST;
```

```
  }

  void set_enter_path()
  {
      int intJudge = 0; /* 用来判断是否决定完成输入 *
/
      char tmpBuf[LARGEST_PATH_NAME]; /** 临时缓冲
区 **/
      while (1)
      {
          printf("Enter The Path You want!\n");
          fgets(tmpBuf, LARGEST_PATH_NAME*sizeof(
char), stdin); /* 获取输入的路径 */
          sscanf(tmpBuf, "%s", to_path_buf);
          if (is_valid_path(to_path_buf) == NOT_E
XIST)
          {
              fprintf(stderr, "Your Enter is Empt
y, So Load the Default Path\n");
              fprintf(stderr, "%s \n", SELF_LOAD_
DEFAULT_PATH);
              strcpy(to_path_buf, SELF_LOAD_DEFAU
LT_PATH);
          }
          fprintf(stdout, "Your Enter is \" %s \"
 ?(1 for yes, 0 for no) \n", to_path_buf);

          fgets(tmpBuf, LARGEST_PATH_NAME*sizeof(
char), stdin);
          sscanf(tmpBuf, "%d", &intJudge); /* 获取
判断数的输入 */
          if (intJudge != 0)
          {
              if (to_path_buf[strlen(to_path_buf)
 - 1] != '/')
                  strcat(to_path_buf, "/");/* 如果
最后一个字符不是'/'，则添加，这里没考虑是否越界 */
              store_hist_path(to_path_buf);
              break;
          } /* if(intJudge) */
```

```
    }/* while (1) */
    return;
}/* set_enter_path */
```

这一组函数的功能稍微复杂，大体来说便是 `读取路径输入->检查路径有效性->读取判断数->是否结束循环`

其中 `_access` 函数有些渊源，因为这个函数被大家所熟知的是这个形式 `access` ，但由于这个形式是 **POSIX** 标准，故 **Windows** 将其实现为 `_access` ，用法上还是一样的，就是名字不同而已。

- 显示历史路径 `show_hist_path`
  - `setPath.h`

```
/**
 * @version  1.0 2015/10/02
 * author    wushengxin
 * function  用于在窗口显示所有的历史路径
 */
void show_hist_path();
```

  - `setPath.c`

```
  void show_hist_path()
  {
      system("cls");
      char outBufName[LARGEST_PATH_NAME] = {'\0'}
;
      FILE* reading = Fopen_s(.file = "PathHistor
y.txt", .mode = "r");
      if (!reading)
      return;

      for (int i = 1; i <= 10 && (!feof(reading))
; ++i)
      {
          fgets(outBufName, LARGEST_PATH_NAME*siz
eof(char), reading);
          fprintf(stdout, "%2d. %s", i, outBufNam
e);
      }
      fclose(reading);
      system("pause");
      return;
  }
```

- 剩下最后一个收尾工作
    - 初始化路径
    - 每次程序启动的时候，我们都会读取本地文件，获取上一次程序使用的最后一个路径，作为当前使用的目的路径
- 初始化目的路径 `init_path`
    - `setPath.h`

```
  /**
   * @versions  1.0 2015/10/02
   * @author    wushengxin
   * @function  用于每次程序启动时初始化目的路径
   */
  void init_path();
```

- `setPath.c`

```
  void init_path()
  {
      int len = 0;
      char last_path[LARGEST_PATH_NAME] = { '\0' };
      FILE* hist_file = Fopen_s(.file = "LastPath.conf", .mode = "r");
      if (!hist_file) /* 打开失败则不初始化 */
          return;
      fgets(last_path, LARGEST_PATH_NAME, hist_file);
      len = strlen(last_path);
      if (len > 1)
      {
          last_path[len - 1] = '\0'; /* 消除一个多余的 ‘\n’ */
          strcpy(to_path_buf, last_path);
      }
      return;
  }
```

这样就大功告成了，对于这个函数中的后 `8` 行代码，没使用惯用的 `fgets 配合 sscanf` 是因为如果这么干的话，需要搭配一个 `memset` 函数清零，后面会有解释。

写在最后方

- 具体思路代码完全都贴出来了，除了主界面的某些细微区别没有贴出来，但是自己应该能够完成。
- 对于 `memset` 的解释
    - 这个函数对于大的内存块的初始化实际上是很慢的，当然我们这个 `30KB` 左右大概的内存可能影响还没有那么大，但是上兆以后，调用 `memset` 就是一种性能问题了，很多情况下，编译器在开启高优化等级之后会自动帮你取消 `memset` 的隐式调用
    - 什么隐式调用，例如 `init_path` 的第二行代码，声明并且用花括号初始化这个数组的时候，就会调用隐式 `memset` 。

结束

- 下一次要实现的就是，本程序的主体 备份

0x0E-单线程备份(下)

写在最前方

- 按部就班的完成一件事情，是十分美妙的感觉。
- 在这里并没有使用 `Makefile` 系列的构造工具，而是使用集成开发环境直接一站式的完成所有的工作，而我们只需要专注于编写有用的代码即可。
- 但是对于这些构造工具的功能还是需要了解的，到了性能瓶颈期，往往是需要这些东西的微调来进行提升，就像算法为什么有那么多的排序算法，看上去复杂度都是一样的，但是快速排序却往往比堆排序要快？不就是因为局部性快速排序要优于堆排序吗？换句话说就是缓存的命中率高
- 不了解底层，永远也无法理解这个解释，但是前方已经有提到过什么叫做空间局部性和时间局部性，至少能有些理解了。
- 构造工具也是如此，例如，编译了源文件生成了库文件，当我们在某个函数中通过该库调用这个库中的某些函数，这个库会在在一开始就加载进我们的程序。当我们的程序十分庞大的时候，也许我们希望在使用的时候才使用它，那么就需要延迟加载这个编译器技术。如果没有了解过构造工具，这些根本不会懂，并且某些情况下**Unix**，**Linux**，**Windows**对于库的加载方式是不同的，这些都是需要了解的，但是我们现在的确没有必要，这个程序满打满算也就是四五百行的代码，不太需要考虑这些。

写在中间

- 上回完成了界面的大部分功能，剩下的便是备份这个主要功能。
- 在完成备份之前，首先想想要如何构造这个备份模型
    - 既然是备份，如果不想扩展为多线程的形式，参考第一次写的遍历函数**(show_structure)**直接找到文件便调用**Windows API(**稍后介绍**)**进行复制即可，不需要讲待备份的文件路径保存下来。
    - 如果要考虑多线程扩展，我们就需要从长计议。
    - 对于一个备份模型，最好的莫过于使用一个队列，依旧实行的是遍历模式，但是将找到的文件路径保存，并放入一个先进先出的队列中，这样我们就能够保证在扩展成多线程的时候，可以有一个很清晰的模型参考。
    - 那么现在的任务就是实现这个用于备份的队列模型。
- 队列模型
    - 应该有一个容器空间：用于存放路径

    - 有队首队尾标志
    - **O(1)**复杂度的检查队列是否为空的接口或标志
    - **O(1)**复杂度的返回容器容量的接口或标志，容器容量应该固定不变
    - 使用一些面向对象的黑魔法，保存一些操作函数防止代码混乱。
        - 初始化函数
        - 释放函数
        - 弹出操作函数
        - 压入操作函数
- 队列实体
    - 考虑到要存储的是字符串，并且由于**Windows API**的参数需求，对于一个文件，我们需要存储的路径有两个**<**源路径，目的路径**>**，对此应该再使用一个路径模型结构体包裹他们，则空间的类型就相应改变一下
- 新建 `Queue.h` `Queue.c`
    - `Queue.h`

```
typedef struct _vector_queue queue;
typedef struct _combine combine;

        |    返回值    | | 函数类型名 ||    参数类型    |
typedef int             (*fpPushBack)(queue * __restrict, const char * __restrict, const char * __restrict);
typedef const combine * (*fpPopFront)(queue *);
typedef void            (*fpDelete)(queue *);
```

五个 `typedef` 不知道有没有眼前一懵。，希望能够很好的理解

前两个是结构体的声明，分别对应着 队列模型 和 路径模型。

后两个是函数指针，作用是放在结构体里，使**C**语言的结构体也能够拥有一些简单的面向对象功能，例如成员函数功能，原理就是可以给这些函数指针类型的变量赋值。稍后例子更加明显。试着解读一下，很简单的。

```
        struct _combine{
        char * src_from_path; /* 源路径 */
        char * dst_to_path;   /* 目的路径 */
        };

        struct _vector_queue{
            combine **        path_contain; /* 存储路径的容器主体 */
            unsigned int      rear;         /* 队尾坐标 */
            unsigned int      front;        /* 队首坐标 */
            int               empty;        /* 是否为空 */
            unsigned int      capcity;      /* 容器的容量 */
            fpPushBack        PushBack;  /* 将元素压入队尾 */
            fpPopFront        PopFront;  /* 将队首出队 */
            fpDelete          Delete;    /* 析构释放整个队列空间 */
        };

        /**
         * @version  1.0 2015/10/03
         * @author   wushengxin
         * @param    object 外部传入的对象指针，相当于 this
         * @function 初始化队列模型，建立队列实体，分配空间，以及设置属性
。

         */
        int newQueue(queue* object);

可以看到，上方的函数指针类型，被用在了结构体内，此处少了一个**初始化函数**，
是因为不打算把他当作**成员函数(借用面向对象术语)**

在使用的时候可以直接`obj_name.PushBack(..., ..., ...);`

更详细的可以看后面的实现部分。成为成员函数的三个函数，将被实现为 `static`
函数，不被外界访问。
```

- `queue.c`

```
int newQueue(queue * object)
{
    queue*      loc_que = object;
    combine**   loc_arr = NULL;

    loc_arr = (combine**)Malloc_s(CAPCITY * sizeof(combine
*));
    if (!loc_arr)
        return 1;

    loc_que->capcity = CAPCITY; /* 容量 */
    loc_que->front = 0;        /* 队首 */
    loc_que->rear = 0;        /* 队尾 */

    loc_que->path_contain = loc_arr; /* 将分配好的空间，放进对
象中 */
    loc_que->PushBack = push_back;
    loc_que->PopFront = pop_front;
    loc_que->Delete   = del_queue;

    return 0;
}
```

在初始化函数中，可以看到，设置了队首队尾以及容量，分配了容器空间，配置了成员函数。

最后三句配置函数的语句中，`push_back`，`pop_front`，`del_queue` 在后方以 `static` 函数实现。

但是由于没有声明，所以切记要将三个 `static` 函数的实现放在 `newQueue` 的前方

```
/**
 * @version  1.0 2015/10/03
 * @author   wushengxin
 * @param    object 外部传入的对象指针 相当于 this
 * @function 释放整个队列实体的空间
 */
static void del_queue(queue * object)
```

```
    {
        Free_s(object->path_contain);
        return;
    }

    /**
     * @version  1.0 2015/10/03
     * @author   wushengxin
     * @param    object 外部传入的对象指针 相当于 this
                 src    源路径
                 dst    目的路径
     * @function 将外部传入的<源路径，目的路径> 存入队列中
     */
    static int push_back(queue * __restrict object, const char
   * __restrict src, const char * __restrict dst)
    {
        int times = 0;
        char*    loc_src = NULL; /* 本地变量，尽量利用寄存器以及缓存
   */
        char*    loc_dst = NULL;
        combine* loc_com = NULL;
        queue*   loc_que = object;

        size_t   len_src = strlen(src); /* 获取路径长度 */
        size_t   len_dst = strlen(dst);
        size_t   rear = loc_que->rear;   /*获取队尾*/
        size_t   front = loc_que->front; /*获取队首*/

        loc_src = Malloc_s(len_src + 1); /* 分配空间 */
        if (!loc_src)
            return 1;

        loc_dst = Malloc_s(len_dst + 1);
        if (!loc_dst)
            return 2;
        strcpy(loc_src, src);
        strcpy(loc_dst, dst);

        loc_com = Malloc_s(sizeof(combine));
        if (!loc_com)
```

```
            return 3;
        loc_com->dst_to_path = loc_dst;
        loc_com->src_from_path = loc_src;

        loc_que->path_contain[rear++] = loc_com; /* 将本地路径加
入实体 */
        loc_que->rear = (rear % CAPCITY);     /* 用数组实现循环队
列的步骤 */

        if (loc_que->rear == loc_que->front)
            loc_que->empty = 0;
        return 0;
    }

    /**
     * @version  1.0 2015/10/03
     * @author   wushengxin
     * @param    object 外部传入的对象指针
     */
    static const combine * pop_front(queue* object)
    {
        size_t   loc_front = object->front;
/*获取当前队首*/
        combine* loc_com   = object->path_contain[loc_front];
/*获取当前文件名*/
        object->path_contain[loc_front] = NULL;     /*出队操作*/
        object->front = ((object->front) + 1) % 20; /*完成出队*/

        if (object->front == object->rear)
            object->empty = 1;
        else
            object->empty = 0;
        return loc_com;
    }
```

一个一个的说这些函数

`del_queue` ：释放函数，直接调用 `Free_s`

`push_back`:压入函数，将外部传入的两个原始的没有组成的路径字符串，组合成一个`combine`，并压入路径，每次都判断并置是否为空标志位，实际上这个函数中有累赘代码的嫌疑，应该再分出一个函数，专门用来分配三个空间，防止这个函数过长(接近**40**行)

`pop_front`:弹出函数，将队列的队首`combine`弹出，用于复制，但是这里有一个隐患，就是要将释放的工作交给外者，如果疏忽大意的话，隐患就是内存泄漏。

没有特地的提供一个接口，用来判断是否为空，因为当编译器一优化，也会将这种接口给优化成直接使用成员的形式，某种形式上的内联。

- 队列模型设计完毕，可以开始设计备份模型
- 备份模型可以回想一下之前的遍历函数，大体的结构一样，只是此处为了扩展成多线程，需要添加一些多线程的调用函数，以及为了规格化，需要添加一个二级界面
- 先设计一下二级界面
- 二级界面
  - 思考一下，这个界面要做什么
    - 选择是否开始备份
    - 并且源路径需要在此处输入
    - 返回上一级
  - 新建 `backup.h` `backup.c` 文件
    - 在主界面选择 **1** 以后就会调用二级界面的函数
    - 列出二级界面的选项
      - **1 Start Back up**
      - **2 Back To last level**
  - `backup.h`

```
/**
 * @version  1.0 2015/10/03
 * @author   wushengxin
 * function  显示二级界面
 */
void sec_main_windows();
```

- `backup.c`

```
  void sec_main_windows()
  {
      char tmpBuf[256];
      int selects;
      do{
          setjmp(select_jmp);
          system("cls");
          puts("-------------------1. Back Up------------------ ");
          puts(" For This Select, You can choose Two Options: ");
          puts(" 1. Start Back up (The Directory Path That You Enter LATER) ");
          puts(" 2. Back To last level ");
          puts("----------------------------------------------------- ");
          fprintf(stdout, "Enter Your Selection: ");
          fgets(tmpBuf, 256, stdin);
          sscanf(tmpBuf, "%d", &selects);
          if (selects != 1 && selects != 2 )
          {
              fprintf(stdout, "\n Your Select \" %s \" is Invalid!\n Try Again \n", tmpBuf);
              longjmp(select_jmp, 1);
          }

          switch (selects)
          {
              jmp_buf enter_path_jmp;
          case 1:
          {
              char tmpBuf[LARGEST_PATH], tmpPath[LARGEST_PATH]; /* 使用栈分配空间，因为只用分配一次 */
              setjmp(enter_path_jmp);         /* enter jump to there */
              puts(" Enter the Full Path You want to BackUp(e.g: C:/Programing/)");
              fprintf(stdout, " Or Enter q to back to sel
```

```
ect\nYour Enter : ");
              fgets(tmpBuf, LARGEST_PATH, stdin);
              sscanf(tmpBuf, "%s", tmpPath);
              if (_access(tmpPath, 0) != 0)   /*检查路径是
否存在，有效*/
              {
                  if (tmpPath[0] == 'q' || tmpPath[0] ==
'Q')
                      longjmp(select_jmp, 0); /* 回到可以
选择返回的界面 */
                  fprintf(stderr, "The Path You Enter is
Not Exit! \n Try Again : ");
                  longjmp(enter_path_jmp, 0); /* enter ju
mp from here */
              }
          }
              break;
          case 2:
              return;
          default:
              break;
          }/* switch */
      } while (1);
      return;
  }
```

这个函数只说几点，首先是 `switch` 的 `case 1` ，之所以用花括号包裹起来的原因是，这样才能在里面定义本地变量，直接在冒号后面定义是编译错误，这个特性可能比较少用，这里提一下，前面也有说过。

写在最后方

- 剩下的就是编写主要的功能函数和线程调用函数了。

0x0F-多线程备份

写在最前方

- 到现在为止我们有了一开始的遍历模型**(show_structure)**，队列模型**(queue)**。
- 现在我们需要做的就是将他们融合在一起，并且通过多线程将其驱动。
- 以下将会用到**Windows API** 和 **Windows**线程库 `<process.h>` 以及文件状态需要用到的 `<sys/stat.h>`
- 对于一个多线程的备份程序而言，可以使用一个十分清晰的方式来实现，通俗的话来说就是，一个线程在不断将路径模型压入队列中，其他 **n**个线程 不断地从这个队列中弹出路径，实行复制 `n = CPU's core * 2 - 1` 。
- 其次我们需要实现的是类似增量备份的效果，即有改变的文件才需要重新复制，或者新增的才需要复制。
- 剩下的就是实现两个供线程调用的函数(这个函数有特殊)，一个入队，一个出队
- 之所以选择**Visual Studio**还有另一个原因，它的某些必要函数是可以开启支持线程安全的，这个概念我不作解释，记得在属性中查看是否开启**/MT(**多线程**)**

写在中间

- 让我们来施展一个很久不用的魔法 `3, 2, 1！`

```
  static queue filesVec;    /* 队列主体 */
  HANDLE vecEmpty, vecFull; /* 两个 Semaphore */
  HANDLE pushThread;  /* 将路径加入队列中的线程 */
  HANDLE copyThread[SELF_THREADS_LIMIT]; /* 将路径弹出队列并复制
的线程 */
  CRITICAL_SECTION inputSec, testSec, manageSec; /* 关键段或临
界区 */

  /* 计算时间 */
  static clock_t start, finish; /* 备份开始，结束时间 */
  static double Total_time;    /* 计算备份花费时间 */
```

这些东西，都被写在了 `backup.c` 中，作为全局变量，暂时先不管其中看不懂的部分，可能到现在为止，大家都已经迷糊了。但是没关系，因为还没说过所以迷糊。继续往下

- 从小事做起，先实现一个简单的增量备份功能
- 实际上就是判断两个文件的最后修改时间是否一致
    - 实现判断目的路径上的文件是否存在
    - 如果存在，则再次判断源路径上的文件和目的路径上的文件的最后修改时间是否相同
- `not_changed`

```
/**
 * @version  1.0 2015/10/03
 * @author   wushengxin
 * @param    dstfile 目的路径的文件
             srcfile 源路径的文件
 * @function 判断两个文件的最后修改时间是否相同
 */
static int not_changed(const char * __restrict dstfile, co
nst char * __restrict srcfile)
{
    struct stat dst_stat, src_stat;
    stat(dstfile, &dst_stat);
    stat(srcfile, &src_stat);
    return dst_stat.st_mtime == src_stat.st_mtime;
}
```

这个函数定义在 `backup.c` 中，因为没有在头文件中声明，所以一定要定义在调用者的前方。

这个函数比较短小，实现的功能就是判断最后修改的时间是否相同，用到头文件 `sys/stat.h`

- 两个被线程调用的函数
    - 首先是入队功能的函数：这个函数主要是调用最后实现的 `backup` 函数，用于递归遍历所给路径下的所有文件夹，将所有文件路径转换成路径模型，压入队中

```
/**
 * @version 1.0 2015/10/03
 * @author  wushengxin
 * @param   pSelect 传入的参数，当前为备份的源路径
 * function 作为线程开始函数的一个参数，作用是调用 backup 函数
 */
static unsigned int __stdcall callBackup(void * pSelect)
{
    char* tmpPath = (char*)pSelect; /* 源路径 */
    start = clock(); /* 开始计时 */
    backup(tmpPath, get_backup_tpath());
    return 0;
}
```

这个函数定义在 `backup.c` 中，因为没有在头文件中声明，所以一定要定义在调用者的前方。

其中参数 `pSelect` 的用法就像是可以接受任何类型的泛型函数，只不过需要自己提前知道类型，这个技术也被用于C语言的面向对象，可用于隐藏成员变量和成员函数。最后可能会稍微介绍一下。

- 其次是出队的函数：这个函数的功能比较多，首先等待队列非空(vecFull)的信号**(Semaphore)**，得到信号之后就弹出一个路径模型，进行复制操作，并且负责把路径模型使用的内存释放，在此释放一个队列空的信号，进入下一个循环。

  ```
  static unsigned int __stdcall callCopyFile(void * para)
  {
      DWORD    isExit   = 0;         /* 判断入队线程是否还存在 */
      queue*   address  = &filesVec;
      combine* localCom = NULL;
      int      empty    = 0;
      while (1)
      {
          char * dst_path = NULL;
  ```

```
        char * src_path = NULL;
        EnterCriticalSection(&testSec);
        GetExitCodeThread(pushThread, &isExit); /*
查看入队的线程是否已经结束 */
        empty = address->empty; /* 查看此时队列是否为
空 */
        LeaveCriticalSection(&testSec);
        if (isExit != STILL_ACTIVE && empty) /* STI
LL_ACTIVE 代表还在运行 */
        {
            puts("Push Thread is End!");
            break;
        }

        isExit = WaitForSingleObject(vecFull, 3000)
; /* 设定一个等待时间，以防死锁 */
        if (isExit == WAIT_TIMEOUT)
        {
            fprintf(stderr, "Copy Thread wait time
out!\n");
            continue;
        }

        EnterCriticalSection(&manageSec); /* 这个关
键段的添加十分重要，是读取时候的核心 */
        if (!(localCom = filesVec.PopFront(address)
)) /* 每次弹出时一定要防止资源争夺带来的冲突 */
            continue;
        LeaveCriticalSection(&manageSec);

        dst_path = localCom->dst_to_path; /* 空间局
部性 */
        src_path = localCom->src_from_path;

        if (CopyFileA(src_path, dst_path, FALSE) ==
 0) /* 显式使用 CopyFileA 函数，而不是使用 CopyFile 宏 */
        {
            EnterCriticalSection(&inputSec);
            if (ERROR_ACCESS_DENIED == GetLastError
())
```

```
            {
                fprintf(stderr, "\nThe File has alr
eady existed and is HIDDEN or ReadOnly! \n");
                fprintf(stderr, "Copy File from %s
Fail!\n", src_path);
            }
            else if (ERROR_ENCRYPTION_FAILED == Get
LastError())
            {
                fprintf(stderr, "\nThe File is Encr
ypted(被加密), And Can't not be copy\n");
                fprintf(stderr, "Copy File from %s
Fail!\n", src_path);
            }
            LeaveCriticalSection(&inputSec);
        }
        Free_s(src_path);
        Free_s(dst_path);
        Free_s(localCom);
        ReleaseSemaphore(vecEmpty, 1, NULL); /* 是
放一个信号量 */
    }/* while (1) */
    return 0;
  }
```

这个函数看似很长，实际上大半实在做判断，而不是在做拷贝，眞正做拷贝的是在中间部分的 `WaitForSingleObject` 函数之后才开始的

- 解释一下
  - 因为在此处并不是多线程的基础文章，而是假设你有基础，如果没有，可以前往一个地方**CSDN**作者： `MoreWindows` ，它的多线程文章十分通俗易懂
  - 这次我们提到的多线程概念有
    - `Semaphore(信号量)` ，使用的一个类似多个互斥量的概念
    - `CRITICAL_SECTION(关键段/临界区)` ，作用和锁相同，但是某些情况下(粗心)不能很好的保护资源不被争夺，不能再进程间共享
    - `Mutex(互斥量)` ，用了非递归的锁一定能保护好资源不被

争夺。但是教 `CRITICAL_SECTION` 的开销要大。

- 其他信息请参看那位的博客。

- 假设你已经具备了多线程的基础。
- 那么讲解一下思路：
  - 首先可以将线程当成这个函数，那么按顺序执行的结果就是，进入循环(好吧废话)
  - 其次我们需要时刻警惕，入队线程是否已经结束？并且结束的话队列是否为空？如果两个条件同时成立，那么就结束本线程，任务结束。
  - 只要任意的条件不符合，就代表本线程的任务还要继续，那么就在原地等待信号，一个队列非空(`vecFull`)的信号。
  - 一旦接受到信号，就证明队列中有路径模型可以被本线程弹出，就开始弹出路径模型，此时一定要记住用关键段或者锁给弹出操作做保险。
  - 这里提一句，互斥量(**Mutex**)比关键段(**Critical Section**)要可靠，但开销更大
  - 弹出之后就是调用**API**进行复制，随后释放堆上的空间，最后释放一个信号，代表队列中的元素被我弹出了一个。
  - 进入下一次循环
- 可以将其中的 `stderr` 换成文件流，将错误信息输入到文件中，而不是屏幕上，以保存错误信息不至于丢失。

- 下面开始主体函数 `backup` 的编写
  - 由于此次的代码过长，所以不放上代码，一切代码都可以到我的**Github**仓库下载。
  - 讲解思路
  - 首先 `backup` 和 `show_structure` 最大的不同便在于后者不需要保存路径模型，而是直接使用。
  - 故我们只需要在 `show_structure` 的路径变量中，添加一个目的路径的参数就行。即 `backup` 函数中的主要参数变为三个:

```
  /* backup.c : backup */
  char * from_path_buf = make_path(path); /* 源路径 */
  char * to_path_buf   = make_path(bpath); /* 目的路径 */
  char * find_path_buf = make_path(path); /* 用于 Windows
API FindFirstFile */
```

- 首先我们拥有一个静态全局的队列 `fileVec` ，可以被任何线程访问
- 紧接着我们构造了两个动作，压入(`backup`)，弹出 (`callCopyFile`)， `backup` 是用 `callBackup` 调用。
- 在二级界面中，当我们选择第一个选项开始备份后，我们选择在此时获得源路径，并将之通过线程创建函数 `_beginthreadex` 传递给 `callBackup` ，进而传递给 `backup` 函数，开始压入任务。

```
/**
 * @version 1.0 2015/10/03
 * @author  wushengxin
 * @param   pSelect 传入的参数，可以是NULL
 * function 作为线程开始函数的一个参数，作用是调用 backup 函数
 */
static unsigned int __stdcall callBackup(void * pSelect)
{
    char* tmpPath = (char*)pSelect;
    start = clock();
    backup(tmpPath, get_backup_topath());
    return 0;
}
```

- 在创建并完成压入线程之后，开始创建拷贝线程，之所以这么安排，是因为压入的操作必定比拷贝的要快，且我们一开始便将信号量的 `vecEmpty` 初始化为 `20` ，这是因为一开始的队列是空的，需要压入线程先开始行动。
- 这里需要提到的是 `_beginthreadex` 函数，还有一个与它相似的函数是 `_beginthread` ，两者之间的区别在于，前者参数更多，前者类似 **POSIX** 里的非分离式线程属性，前者使用完需要手动销魂，前后者调用的函数修饰不一样，什么意思？如果下面这个代码使用后者创建会发生什么问题？
- 想想分离式线程的特点，就是自动释放所有的资源，这就会导致，如果前一个线程比自己创建的还快完成任务，那么自己就可能用到它的句柄，这就可能会造成错误。而如果前者的话，由程序员稍后自己释放销毁句柄，能保证一定不会出现这种现象。

- 一直以来都是使用前者。

```
  /* backup.c : backup */
  pushThread = (HANDLE)_beginthreadex(NULL, 0, callBackup
, (void*)tmpPath, 0, NULL); /* 压入线程 */
  for (int i = 0; i < SELF_THREADS_LIMIT; ++i)
  {
      copyThread[i] = (HANDLE)_beginthreadex(NULL, 0, cal
lCopyFile, NULL, 0, NULL); /* 拷贝线程 */
  }
```

- 在压入的过程中，唯一需要注意的就是在压入 `fileVec` 的时候，一定要防止资源竞争(同样适用在复制过程中的弹出操作)，通过信号量可以有效防止多于1个以上的线程同时访问 `fileVec`

```
  /* backup.c : backup */
  if(is Directory)
      { ... }
  else /* 是一个文件 */
  {
      strcat(tmp_from_file_buf, fileData.cFileName);
      strcat(tmp_to_file_buf, fileData.cFileName);
      if (_access(tmp_to_file_buf, 0) == 0) /*如果目标文件
存在*/
      {
          if (is_changed(tmp_from_file_buf, tmp_to_file_b
uf))
          {
              rele_path(tmp_from_file_buf);
              rele_path(tmp_to_file_buf);
              continue;  /*如果目标文件与源文件的修改时间相同
，则不需要入队列*/
          }
          fprintf(stderr, "File : %s hast changed!\n", tm
p_from_file_buf);
      }
      else
          fprintf(stderr, "Add New File %s \n", tmp_from_
file_buf);
      /* 使用信号量防止竞争 */
      WaitForSingleObject(vecEmpty, INFINITE);
      EnterCriticalSection(&manageSec);
      filesVec.PushBack(&filesVec, tmp_from_file_buf, tmp
_to_file_buf);
      LeaveCriticalSection(&manageSec);
      ReleaseSemaphore(vecFull, 1, NULL);
  }
```

- 在复制的过程中，十分有可能出现压入线程结束，但是拷贝线程却停留在等待信号的阶段，这就要求我们必须设定一个等待的时间，超时则重新检测是否是压入线程结束且队列空。这一点十分重要，可以自己思考一下。

```
  /* backup.c : callCopyFile */
  EnterCriticalSection(&testSec);
  GetExitCodeThread(pushThread, &isExit); /* 查看入队的线程是否已经结束 */
  empty = address->empty; /* 查看此时队列是否为空 */
  LeaveCriticalSection(&testSec);
  if (isExit != STILL_ACTIVE && empty) /* STILL_ACTIVE 代表还在运行 */
  {
      puts("Push Thread is End!\n");
      break;
  }

  isExit = WaitForSingleObject(vecFull, 3000); /* 设定一个等待时间，以防死锁 */
  if (isExit == WAIT_TIMEOUT)
  {
      fprintf(stderr, "Copy Thread wait time out!\n");
      continue; /* 所有代码都在一个 while(1)中 */
  }
```

  - 当所有线程都退出就代表任务完成，要销毁一系列相关参数。

写在最后

- 添加了多线程以后，前方有一些原始代码是需要修改的才能使用，比如队列模型(`Queue.c`)中的一些代码，需要用关键段进行修饰，防止资源争夺。其他方面没有太多需要修改的
- 完整代码被我放在我的**Github**仓库

简单总结

- 使用的 **Windows API** 中 `CopyFile` `CreateDirectory` `FindFirstFile` `FindNextFile` ，是核心的功能函数。
- 在此处，可以换一个思路思考一下，是否可以对容器队列，进行线程安全保护，从而不必在主代码中一直使用关键段进行保护？至少在 `PushBack` 和 `PopFront` 两个操作上可以不必担心资源争夺。防止在编写程序的时候粗心大意忘记了保护。

总结

- 首先前面提到了一个思路：给队列模型添加初步的线程保护，在使用它的时候，可以不考虑会保护其免受资源争夺的问题
    - 实际上就是将 `CRITICAL_SECTION` 放在 `Queue.c` 的实现当中。
    - 让两个基本操作 `PushBack` `PopFront` 能够自己实现保护自己
    - 具体应该怎么做呢？之前的我们对队列模型中的 `empty` 实现了单独保护，现在反过来，将其保护范围扩大一些就行了。
- 具体方法 `Queue.c`
    - 首先是取消使用 `empty_sec` 这个关键段/临界区
    - 使用新的 `static CRITICAL_SECTION io_section;`
    - 修改 `newQueue` 和 `del_queue` 里的初始化和销毁关键段代码。
    - 以及重点的 `push_back` 和 `pop_front` 的代码修改，前者变化多一些

    ```
      static int push_back(queue * __restrict object, const char * __restrict src, const char * __restrict dst)
      {
          char*    loc_src = NULL; /* 本地变量，尽量利用寄存器以及缓存 */
          char*    loc_dst = NULL;
          combine* loc_com = NULL;
          queue*   loc_que = object;

          size_t   len_src = strlen(src); /* 获取路径长度 */
          size_t   len_dst = strlen(dst);
          size_t   rear = 0; /* 队列的队尾 */
          size_t   front = 0; /* 队列的队首 */

          loc_src = Malloc_s(len_src + 1); /* 分配空间 */
          loc_dst = Malloc_s(len_dst + 1);
          loc_com = Malloc_s(sizeof(combine));
          if (loc_src == NULL || loc_dst == NULL || loc_com == NULL)
          {
              Free_s(loc_src); /* 特殊处理过的释放函数 */
              Free_s(loc_dst);
              Free_s(loc_src);
    ```

```
            return 1;
        }
        strcpy(loc_src, src); /* 构造路径模型 */
        strcpy(loc_dst, dst);
        loc_com->dst_to_path = loc_dst;
        loc_com->src_from_path = loc_src;
        /* 进入保护 */
        EnterCriticalSection(&io_section);
        rear = loc_que->rear;   /*获取队尾*/
        front = loc_que->front; /*获取队首*/

        loc_que->path_contain[rear++] = loc_com; /* 将本地路
径加入实体 */
        loc_que->rear = (rear % CAPCITY);     /* 用数组实现循
环队列的步骤 */
        /* 取消原先的保护 */
        if (loc_que->rear == loc_que->front)
        {
            loc_que->empty = 0;
        }
        LeaveCriticalSection(&io_section);
        return 0;
    }
```

注释里写了很多信息，主要教之前的版本改变了一下串行代码的顺序，功能并没有太大变化，变化的两处地方，一个是内存分配错误判断由三个 `if` 变成一个 `if`，另一个是为了使临界区内的代码尽可能少，所以将一些操作移动了。

`pop_front` 代码基本没改变只是将临界区扩大了保护范围。

```
static combine * pop_front(queue* object)
{
    EnterCriticalSection(&io_section);
    size_t   loc_front = object->front;
 /*获取当前队首*/
    ...
//    EnterCriticalSection(&empty_sec); /* 原先的临界区起始 */
    if (object->front == object->rear)
        object->empty = 1;
    else
        object->empty = 0;
//    LeaveCriticalSection(&empty_sec);
    LeaveCriticalSection(&io_section);
    return loc_com;
}
```

- 如此修改以后，该队列模型就具备了初步的线程安全功能。
- 在主代码中，可以删除 `PopFront` 和 `PushBack` 附近的保护操作。
- 前方提到了， `CRITICAL_SECTION` 相对于 `Mutex` 不太安全，这里简单说一下，具体请查询相关资料
  - 前者只对当前代码段负责，也就是其他操作这个资源的途径是不被保护的。
  - 通俗的来说，假设有多个线程， `CRITICAL_SECTION` 只保证在同一个"时间"内，只有一个线程能够运行这段代码，假设我在其他代码还有对这段代码中的资源进行访问，那关键段就不能保证什么了。
  - 速度快开销小是因为它和内核关系不大，是进程内的操作。
- 发现问题了吗？
  - 在代码中，对于**empty** 的操作存在着问题，我们必须对它进行类似 `Mutex` 的保护，而不是使用 `CRITICAL_SECTION`
  - 我们这里应该使用 `Mutex` 吗？其实有一个更好的选择，那就是在**Windows Vista**之后引入的一个读写锁 `SRWLOCK` ，允许多个线程读取数据或者单个线程写入数据
    - 为什么选择它？道理还是一样，因为它不使用内核资源。
    - 将代码中对 `empty` 的关键段保护修改或添加上 `SRWLOCK` 读写锁的保护

- 操作并没有什么区别，就是进入保护区请求(`AcquireSRWLock(Exclusive/Shared)`)，离开保护区释放(`ReleaseSRWLock(Exclusive/Shared)`)。
- 本来有一个更好的可以减小开销的 `TryAcquire...` 操作，但是确在**Windows 7**以后才引入，故不在此实现。

结后语

- 所谓读写锁，并不是所谓的银弹，意思就是不要盲目的使用读写锁，这里使用读写锁只是因为想要更加全面的覆盖知识点！
- 要知道读锁的加持，并不一定就比一个互斥锁(这是**Linux**平台下对**Windows**临界区的称呼)要廉价，特别是在库设计实现者手里，他必须考虑到种种因素，例如 写锁饿死 现象，想要解决这个问题，就不可避免的要牺牲读锁的性能，有可能将互斥锁替换成读写锁以后，性能反而降低了。

> 写锁饿死，是个挺广泛的概念，只要有读写锁的身影就必定有它，可以查阅相关资料

- 这一切都是需要测试，测试，再测试，之所以选择在 **Windows** 平台上开发，原因之一就是**Visual Studio**有一套强大到爆表的性能分析工具，你可以轻易找出代码性能问题。善用它来研究不同锁之间的差别，性能。

- 对于多线程程序而言，同步原语实际上还是要善用 条件变量/互斥锁(临界区,关键段)这两个概念，能满足90% 的需求，最多再使用一些平台关键字就行了，不要参杂着各种各样的同步魔法，要不就换一种思维，除了 多线程，并发世界还有许多“很美好”的东西

> 很美好。。。是我自己说的

第四部分

> 所谓系统编程之一
>
> TCP/IP 编程
>
> 常见网络协议及UDP, TCP 应用。
>
> 最后以详述描写，如何在Linux上编写一个并发HTTP服务器作为本章结尾。

0x10-网络的世界

写在最前方

- 网络编程没有想象之中的难，但是同样一句废话，也没有想象之后那么容易。
- 接下来记录的是对于网络编程的一些教接近底层的东西，也就是称之为系统接口函数的东西，通常叫做系统编程，
- 当然网络编程在非学院派看来，是使用一些成熟的库(这是对于C语言来说，当然很少有人愿意这么做，但个人觉得有了库的C就和其他高级语言更像了)（注：C/C++都没有标准网络库，所以只能使用第三方开发的库，所谓乱世出英雄。C++在 C++17 似乎要有了。），例如 `libev` 这一类的。
- 最后，还是先将底层基础打好为妙。

> 开始首先是万物根源的协议信息

概念

- 最具误导性的当属于 `TCP/IP` 协议了
  - 所谓 `TCP/IP` 协议指的并不是一个协议，往往在生活中听见的术语如：**IP**地址， **TCP**连接 等，总会被误导，以为就是一个东西
  - 实际上它们都是彼此独立的 协议 ，只不过会相互合作罢了
  - `TCP/IP` 说的是一个 协议族 ，也就是说是一堆协议的统称
- 对比 **OSI** 和 **TCP/IP** 参考模型：

OSI	TCP/IP
应用层 表示层 会话层	应用层
传输层	传输层
网络层	网络层
链路层 物理层	网络接口层

- 其中最常接触的
  - 位于 网络层 的 **IP** 协议，大家所熟知的 `IP地址` 就是由它进行封装并传往下一层
  - 位于 传输层 的 **TCP/UDP** 两个协议， 一个是面向连接(STREAM), 一个是面向数据(DGRAM)的，实际上还有一个但这里不记录。

    - 查看自身 网络信息的办法
        - `*nix`：在 **Terminal** 中输入 `ifconfig -a`
        - `Windows`：在 **PowerShell** 中输入 `ipconfig`
- 概念模糊的 **DNS**
    - 其实很简单，它的作用就是用来找到域名所对应的 **IP**地址
    - 为什么？因为 **IP**地址 太难记了！如果你觉得 **IPv4** 地址还难不倒你，那请你试试 **IPv6**
    - 怎么查看域名对应的 **IP**地址，当然先不考虑 **CDN**
        - `*nix` 和 `Windows` 都可以通过 `ping <domain name>` 命令进行查询
- **MAC**地址 和 端口号
    - 对于前者，实际上应该是最熟悉不过的，对于网络上的主机而言，每一台主机就有一个专属的 **MAC**地址
    - 后者则是相当于一个房子的门，这个比喻在各大教材中广泛引用，但也的确贴切，假设 **IP**地址 是房子的地址，那么到了别人家要知道门在哪才行。

一个完整的应用程序传输数据时候 封装 的过程(从右二向左依次封装)：

以太网首部	**IP**	**TCP/UDP**	真实数据	尾部
MAC地址	IP地址	TCP或者UDP协议	应用程序数据	效验码
源和目的MAC地址以及	及前层协议类型	源和目的端口号及前层应用程序首部信息	应用软件信息和真正的数据	

其中端口号实际上就是 应用程序的信息

接收数据时的 拆解 顺序与 封装 正好相反。

- 其中在传输过程中，作为接收方最开始使用的是 网络接口层/数据链路层 的驱动程序(即操作系统自带或另行安装，总之不用使用的程序员写就对了)，来判断这个包是否属于我，判断的依据就是 **MAC**地址，如果是再判断什么协议
    - 在此处的协议可不止 **IP**协议， 也可能是 **ARP**协议 等。之后就是就事论事交给相应的处理软件去处理(拆解)就行
    - 科普： MAC地址是 `48bit` 的， 前 `24bit` 由 **IEEE** 分配， 后 `24bit`

由厂商分配。原则上是唯一的。

- **MAC**地址 和 **IP**地址
  - 既然前方说到 **MAC**地址 和 **IP**地址 都能够作为识别另一个主机的唯一标识，但是为什么需要有两个相同功能的东西？
  - 是，在一开始，网络很小的情况下，例如我们在同一个局域网中，我们之间需要通信的时候，只需要使用ARP协议，进行广播，向在一个网络中的所有主机发送消息就行，剩下的就让其他主机去判断(通过**MAC**地址)这个数据是不是发给我的。
    - **ARP**协议 的作用就是在同一个网络中，通过 广播 找出符合自己要求的主机的 **MAC**地址 ，如果不在同一个网络中，又想知道对方的 **MAC**地址， 那只能借助把每个网络链接在一起的 网关 来帮助你发送 。 总之进行网络通信时必须知道对方的 **IP**地址 和 **MAC**地址
  - 但是如果是现在整个互联网呢？不算 **IPv6** ，就算 **IPv4** 也是几十亿的存在，如果我从中国向国外发送信息，广播整个互联网的所有主机，那就炸了！
  - 所以我们需要对世界网络进行分区，让大区域包含小区域，就像国家**-**省**-**市区**...** ， 很遗憾的是 **MAC**地址 是跟计算机相关而不是和位置相关的。所以我们有了 **IP**协议
  - **IP**协议 所附带的产品 **IP**地址 的作用就在帮助计算机识别自己是否在同一个网络中(这里省略了子网掩码的作用)。
- 实际上，在进行网络编程的时候，以上细节几乎都被隐藏起来，留给我们的只是可供使用的接口。

> 也许，许多大学计算机基础课程，会讲到 IP地址 有种类，分为 A,B,C...类，老师还介绍了各种类型的地址范围。
>
> 但是在现代，这种分类早已经失效，或者说正在逐渐消失，因为当下的 IP 地址的 子网掩码 可以是任意位，并以反斜杠跟在 IP地址后方。
>
> 比较现代的 IP地址 表示形式一般如此 1.185.223.1/24 代表着子网掩码是由 24 个 从左至右连续的的二进制1 组合而成，其余位为0。称为**CIDR**分类

夹在中间

> 事实上有一些实用且挺炫酷的函数，可以先提一下

- 域名 和 IP地址 的互查

- `gethostbyname` 用于域名查找IP信息及各类信息
    - `struct hostent * gethostbyname(const char * hostname)`
    - `struct hostent` 是存储查找到的各类型信息，后方会有介绍
    - `hostname` 即要查询的域名
- `gethostbyaddr` 用于IP地址查找 域名及各类信息
    - `struct hostent * gethostbyaddr(const char * addr, socklen_t len, int family)`
        - `addr` 是要查询的 IP地址，之所以是 `const char *` 是因为C语言历史遗留的原因，实际上其类型应为 `struct in_addr *` (IPv4)
        - `len` 地址的长度，即 **IPv4** 为4， **IPv6** 为16
        - `family` 即协议的种类， **IPv4** 为 `AF_INET` , **IPv6** 为 `AF_INET6`

struct hostent 的成员	.	类型	.	解释
h_name		char *		官方名称
h_aliases		char **		域名集合，以NULL结尾
h_addrtype		int		地址族的类型 AF_INET 或 AF_INET6
h_length		int		地址的长度 4 或 16
h_addr_list		char **		IP的集合，以NULL结尾, 实际上每个元素的类型为 struct in_addr*

- 其中第二和最后一个是关注的重点所在，可以在调用函数之后，输出信息

    > 实际上，这并不是一个好的方法，在后方将记录 现代人的我们 该如何做到这些事情，以上只是以前的TCP/IP 编程

> 只适用于 IPv4

套接字网络编程初始

> 选择使用 C 语言进行编程

- 在网络编程中，最常实用的两种连接方式 `TCP` 和 `UDP`
- 最常编程的平台 `POSIX 标准->*nix平台标准` 和 `Windows 平台标准`

- 实际上，后者也是参考前者进行一些细微的改变(指的是接口)

> 对比两种不同连接方式的不同地位的创建，使用

TCP服务器	TCP客户端	UDP服务器	UDP客户端	注释
socket()	socket()	socket()	socket()	创建套接字
bind()		bind()	bind()	绑定所分配IP地址和端口号
listen()	connect()			客户端则绑定IP地址和端口号，并等待连接；服务器则是等待连接
accept()				服务器接受连接
...	...	sendto/recvfrom()	sendto/recvfrom()	对于UDP即是连接也是操作
close()	close()	close()	close	双向直接关闭连接
shutdown()	shutdown()	shutdown()	shutdown()	可选择方向的关闭连接,即更加灵活

如此对比虽然有一些小瑕疵，但是能够大体上反映出眞个网络编程上不同方式的区别

> 注1： 对于 sendto recvfrom 这两个接口函数，并不一定是只能用在 UDP类型的 套接字上，同样 TCP类型的 套接字也能使用，但是这么做并没有什么意义。
>
> 注2： 实际上 UDP 没有所谓的 服务器和和护短，因为本来就是单纯的互相发来发去。客户端端口 一般是随机的
>
> 以上是 *nix平台下的标准， Windows下的操作方式和 API有细微不同，但大部分是一致的。

Windows	*nix
socket()	socket()
bind()	bind()
connect()	connect()
listen()	listen()
accept()	accept()
closesocket()	close()
send()	send()
read()	read()
sendto()	sendto()
recvfrom()	recvfrom()

不仅仅是接口名字相同，参数个数以及功能也是一致，即使有一个例外，其参数以及使用方法也相同。

> 那岂不是可以直接移植了？
>
> 并不！

在 **Windows** 套接字编程时 ， 由于 `Windows` 将其实现为动态库，所以在使用时需要将其加载进程序。

故而多加了加载操作。

```
int WSAStartup(
  WORD      wVersionRequested,
  LPWSADATA lpWSAData  /* 这是一个结构体， 传入类型为WSADATA*  */
);
int WSACleanup(void);
```

每当在 **Windows** 上进行套接字编程时，总要指定某个版本的套接字库：

```
WSADATA wsaData;
int err_code;
/*
* MAKEWORD()的作用在于将版本号转为指定格式传入
* 当下(2015-10)套接字库的版本号最高是 2.2
*/
err_code = WSAStartup(MAKEWORD(2, 2), &wsaData);
/* TODO Something */
WSACleanup();
```

这是最基本的在 **Windows** 上使用 套接字 编程的流程，但是如果本平台的套接字库最高版本并不符合当前要求呢？

那么首先会将套接字版本库尽可能设置到平台的 最高版本 ，可以通过结构体 `WSADATA` 进行查询

```
if (LOBYTE(wsaData.wVersion) != 2 || HIBYTE(wsaData.wVersion) !=
 2)
{
  printf("Could not find a usable version of Winsock.dll\n");
  WSACleanup();
  return 1;
}
```

> 总体而言， `Windows平台` 和 `*uix平台` 的区别在于，前者使用时需要 加载 和清除 套接字库 其余逻辑流程一致，毕竟只有统一才能越利于编程世界的发展。

0x11-套接字编程-1

套接字编程

- 两种协议 `TCP` 和 `UDP`
    - 前者可以理解为有保证的连接，后者是追求快速的连接
    - 当然最后一点有些 太过绝对 ，但是现在不需熬考虑太多，因为初入套接字编程，一切从简
    - 稍微试想便能够大致理解， `TCP` 追求的是可靠的传输数据， `UDP` 追求的则是快速的传输数据
    - 前者有繁琐的连接过程，后者则是根本不建立可靠连接(不是绝对)，只是将数据发送而不考虑是否到达。

> 以下例子以 `*nix` 平台的便准为例，因为 `Windows` 平台需要考虑额外的加载问题，稍作添加就能在 Windows 平台上运行

UDP

- `UDP`
    - 这是一个十分简洁的连接方式，假设有两台主机进行通信，一台只发送，一台只接收。
    - 接收端：

```
    int sock; /* 套接字 */
    socklen_t addr_len; /* 发送端的地址长度，用于 recvfrom *
/
    char mess[15];
    char get_mess[GET_MAX]; /* 后续版本使用 */
    struct sockaddr_in recv_host, send_host;

    /* 创建套接字 */
    sock = socket(PF_INET, SOCK_DGRAM, 0);

    /* 把IP 和 端口号信息绑定在套接字上 */
    memset(&recv_host, 0, sizeof(recv_host));
    recv_host.sin_family = AF_INET;
    recv_host.sin_addr.s_addr = htonl(INADDR_ANY);/* 接收
任意的IP */
    recv_host.sin_port = htons(6000); /* 使用6000 端口号 *
/
    bind(sock, (struct sockaddr *)&recv_host, sizeof(recv
_host));

    /* 进入接收信息的状态 */
    recvfrom(sock, mess, 15, 0, (struct sockaddr *)&send_
host, &addr_len);

    /* 接收完成，关闭套接字 */
    close(sock);
```

上述代码省略了许多必要的 错误检查 ，在实际编写时要添加

- 代码解释：
    1. **PF_INET** 代表协议的类型，此处代表 **IPv4** 网络协议族， 同样 **PF_INET6** 代表 **IPv6** 网络协议族，这个范围在后方单独记录，不与IPv4混在一起(并不意味着更复杂，实际上更简便)。
    2. **AF_INET** 代表地址的类型，此处代表 **IPv4** 网络协议使用的地址族，同样有 **AF_INET6** (在操作系统实现中 PF_INET 和 AF_INET 的值一样，但是还是要写宏更好，而不应该直接用数字或者，混淆使用)
    3. `htonl` 和 `htons` 两个函数的使用涉及到 大端小端问题， 这里不叙述，需要记住的是在网络编程时一定要使用这种函数将必要信息转

为 大端表示法 。

4. `(struct sockaddr *)` 这个强制转换是为了参数的必须，但不会出错，因为 `sizeof(struct sockaddr_in) == sizeof(struct sockaddr)` 具体可以查询相关信息，之所以这么做是为了方便编写套接字程序的程序员。

- 发送端：

```
    int sock;
    const char* mess = "Hello Server!";
    char get_mess[GET_MAX]; /* 后续版本使用 */
    struct sockaddr_in recv_host;
    socklen_t addr_len;
    /* 创建套接字 */
    sock = socket(PF_INET, SOCK_DGRAM, 0);
    /* 绑定 */
    memset(&recv_host, 0, sizeof(recv_host));
    recv_host.sin_family = AF_INET;
    recv_host.sin_addr.s_addr = inet_addr("127.0.0.1");
    recv_host.sin_port = htons(6000);
    /* 发送信息 */
    /* 在此处，发送端的IP地址和端口号等各类信息，随着这个函数的调用，自动绑定在了套接字上 */
    sendto(sock, mess, strlen(mess), 0, (struct sockaddr *)&recv_host, sizeof(recv_host));
    /* 完成，关闭 */
    close(sock);
```

上述代码是发送端。

- 代码解释：

1. `inet_addr` 函数是用于将字符串格式的 **IP**地址 转换为 大端表示法 的 地址类型，即 `s_addr` 的类型 `in_addr_t`
2. 与之相反，同样也有功能相反的函数 `inet_ntoa` 用于将 `in_addr_t` 类型转为字符串，但是使用时一定要记住及时拷贝返回值

```
char addr[16];
recv_host.sin_addr.s_addr = inet_addr("127.0.0.1");
strcpy(addr, inet_ntoa(recv_host.sin_addr.s_addr));
```

- 从上述代码看出， `UDP` 协议的使用十分简洁，几乎就是 创建套接字**->**准备数据**->**装备套接字**->**发送**/**接收**->**结束
- 其中，都没有连接的操作，但是实际上这是为了方便 `UDP` 随时和 不同的主机 进行通信所默认的设置，如果需要和相同主机一直通信呢？
- 此中的原由暂时不需要知道，记录方法，即长时间使用 `UDP` 和同一主机通信时，可以使用 `connect` 函数来进行优化自身。此时 假设两台主机的实际功能一致，既接收也发送
- 发送端：

```
    /* 前方高度一致，将 bind函数替换为 */
    connect(sock, (struct sockaddr *)&recv_host, sizeof(recv_host); // 将对方的 IP地址和 端口号信息 注册进UDP的套接字中)
    while(1) /* 循环的发送和接收信息 */
    {
      size_t read_len = 0;
      /* 原先使用的 sendto 函数，先择改为使用 write 函数， Windows平台为 send 函数 */
      write(sock, mess, strlen(mess));            /* send(sock, mess, strlen(mess), 0) FOR Windows Platform */
      read_len = read(sock, get_mess, GET_MAX-1); /* recv(sock, mess, strlen(mess)-1, 0) FOR Windows Platform */
      get_mess[read_len-1] = '\0';
      printf("In Client like Host Recvive From Other Host : %s\n", get_mess);
    }
    /* 后方高度一致 */
```

- 接收端：

```
    /* 前方一致， 添加额外的 struct sockaddr_in send_host;
并添加循环，构造收发的现象*/
        while(1)
        {
          size_t read_len = 0;
          char sent_mess[15] = "Hello Sender!"; /* 用于发
送的信息      */
          sendto(sock, mess, strlen(sent_mess), 0, (struc
t sockaddr *)&recv_host, sizeof(recv_host));
          read_len = recvfrom(sock, mess, 15, 0, (struct
sockaddr *)&send_host, &addr_len)
          mess[read_len-1] = '\0';
          printf("In Sever like Host Recvive From other H
ost : %s\n", mess);
        }
        /* 后方高度一致 */
        /*
        * 之所以只在接收端使用 connect 的原因，便在于我们模拟的
是 客户端-服务器 的模型，而服务器的各项信息是不会随意变更的
        * 但是 客户端就不同了，可能由于 ISP(Internet Server P
rovider) 的原因，你的IP地址不可能总是固定的，所以只能
        * 保证 在客户端 部分注册了 服务器 的各类信息，而不能在 服
务器端 注册 客户端 的信息。
        * 当然也有例外，例如你就想这个软件作为私密软件，仅供两个人
使用， 且你有固定的 IP地址，那么你可以两边都connect，但是
        * 一定要注意，只要有一点信息变动，这个软件就可能无法正常的
收发信息了。
        */
```

- 代码解释
    - 故而实际上，虽然前方的表格显示，`UDP` 似乎并没有 `connect` 的使用必要，但是实际上还是有用到的地方。
    - 就 `*nix` 的 **API** 来说，`sendto` 和 `write` 的区别十分明显，便是一个需要在参数中提供目标主机的各类信息，而后者则不需要提供。同样的道理 `recvfrom` 和 `read` 也是如此。
    - 这个代码只是做演示而已，所以将代码置于无限循环当中，现实中可以自行定义出口条件。

> 以上是 UDP 的一些简单说明，入门足矣，并未详细叙述某些 函数 的具体用法，而是用实际例子来体现。 在 记录 TCP 之前，还是需要讲一个函数 shutdown

- `shutdown` 与 `close(closesocket)`
  - 首先要知道，网络通信一般而言是双方的共同进行的，换而言之就是双向的，一个方向只用来发送消息，一个方向只用来读取消息。
  - 这就导致了，在结束套接字通信的时候，需要关闭两个方向的通道(暂时叫它们通道)，那同时关闭不行吗？可以啊
    - `close(sock); // closesocket(sock); FOR Windows PlatForm` 就是这么干的，同时断开两个方向的连接。
    - 简单的通信程序或者单向通信程序这么做的确无甚大碍，但是万一在结束通信的时候需要接收最后一个信息那该怎么办？
      - 假设通信结束，客户端向服务器发送 "Thank you"
      - 服务器需要接收这个信息，之后才能关闭通信
      - 问题就在这之间，服务器并不知道客户端会在通信结束后的什么时刻传来信息
      - 所以我们选择在通信完成后先关闭 服务器的 发送通道(写流)，等待客户端发来消息后，关闭剩下的 接收通道(读流)
  - 发送端：

```
    /* 假设有一个 TCP 的连接，此为客户端 */
    write(sock, "Thank you", 10);
    close(sock); // 写完直接关闭通信
```

  - 接收端：

```
    /* 此为服务器 */
    /* 首先关闭写流 */
    shutdown(sock_c, SHUT_WR);
    read(sock_c, get_mess, GET_MAX);
    printf("Message : %s\n", get_mess);
    close(sock_c);
    close(sock_s); // 关闭两个套接字是因为 TCP 服务器端的需要
，后续会记录
```

- 代码解释
  - `shutdown` 函数的作用就是 可选择的关闭那个方向的输出
    - `int shutdown(int sock, int howto);`
    - `sock` 代表要操作的套接字
    - `howto` 有几个选择
      - ***nix**：`SHUT_RD` `SHUT_WR` `SHUT_RDWR`
      - **Windows**：`SD_RECEIVE` `SD_SEND` `SD_BOTH`

停下来

1. 程序员应该越来越来，做的事情应该越来越少，但是能达到的成就应该越来越多
2. 在 IPv6 出现的今天，网络编程已经开始向简洁和强大靠近，即便是身为底层语言的 C语言
3. 实际上由于 C语言 并没有自己的网络库， 故为了能进行网络编程，不得不依赖于系统函数，这就是所谓的系统编程， 你已经是一个系统程序员了。
4. 而 系统函数 随着时代的变化，正在不断完善，增加(几乎没有废除的先例，所以不用担心之前的程序无法运行)。
5. 相应的，由于以前的网络编程只适合于 IPv4 的地址，自从出现了 IPv6, 我们需要一套全新的方式，正好他来了。

0x12-套接字编程-2

新时代的 套接字网络编程

1. 首先有几个结构体，以及一个接口十分重要及常用：
    - `struct sockaddr_in6` ：代表的是 IPv6 的地址信息
    - `struct addrinfo` ：这是一个通用的结构体，里面可以存储 IPv4 或 IPv6 类型地址的信息
    - `getaddrinfo` ：这是一个十分方便的接口，在上述 UDP 程序中许多手动填写的部分，都能够省去，有该函数替我们完成
2. 改写一下上方的例子：

    - 接收端：

```
	int sock; /* 套接字 */
	socklen_t addr_len; /* 发送端的地址长度，用于 recvfrom */
	char mess[15];
	char get_mess[GET_MAX]; /* 后续版本使用 */
	struct sockaddr_in host_v4; /* IPv4 地址 */
	struct sockaddr_in6 host_v6; /* IPv6 地址 */
	struct addrinfo easy_to_use; /* 用于设定要获取的信息以及如何获取信息 */
	struct addrinfo *result;    /* 用于存储得到的信息(需要注意内存泄露) */
	struct addrinfo * p;

	/* 准备信息 */
	memset(&easy_to_use, 0, sizeof easy_to_use);
	easy_to_use.ai_family = AF_UNSPEC; /* 告诉接口，我现在还不知道地址类型 */
	easy_to_use.ai_flags = AI_PASSIVE; /* 告诉接口，稍后“你”帮我填写我没明确指定的信息 */
	easy_to_use.ai_socktype = SOCK_DGRAM; /* UDP 的套接字 */
	/* 其余位都为 0 */

	/* 使用 getaddrinfo 接口 */
```

```
    getaddrinfo(NULL, argv[1], &easy_to_use, &result); /
* argv[1] 中存放字符串形式的 端口号 */

    /* 创建套接字，此处会产生两种写法，但更保险，可靠的写法是如此
 */
    /* 旧式方法
    *  sock = socket(PF_INET, SOCK_DGRAM, 0);
    */
    /* 把IP 和 端口号信息绑定在套接字上 */
    /* 旧式方法
    *  memset(&recv_host, 0, sizeof(recv_host));
    *  recv_host.sin_family = AF_INET;
    *  recv_host.sin_addr.s_addr = htonl(INADDR_ANY);/*
接收任意的IP */
    *  recv_host.sin_port = htons(6000); /* 使用6000 端口
号 */
    *  bind(sock, (struct sockaddr *)&recv_host, sizeof(
recv_host));
    */

    for(p = result; p != NULL; p = p->ai_next) /* 该语法
需要开启 -std=gnu99 标准*/
    {
      sock = socket(p->ai_family, p->ai_socktype, p->ai_
protocol);
      if(sock == -1)
        continue;
      if(bind(sock, p->ai_addr, p->ai_addrlen) == -1)
      {
        close(sock);
        continue;
      }
      break; /* 如果能执行到此，证明建立套接字成功，套接字绑定
成功，故不必再尝试。 */
    }

    /* 进入接收信息的状态 */
    //recvfrom(sock, mess, 15, 0, (struct sockaddr *)&se
nd_host, &addr_len);
    switch(p->ai_socktype)
```

```
    {
      case AF_INET :
        addr_len = sizeof host_v4;
        recvfrom(sock, mess, 15, 0, (struct sockaddr *)&
host_v4, &addr_len);
        break;
      case AF_INET6:
        addr_len = sizeof host_v6
        recvfrom(sock, mess, 15, 0, (struct sockaddr *)&
host_v6, &addr_len);
        break;
      default:
        break;
    }
    freeaddrinfo(result); /* 释放这个空间，由getaddrinfo分
配的 */
    /* 接收完成，关闭套接字 */
    close(sock);
```

- 代码解释：
  - 首先解释几个新的结构体
    1. `struct addrinfo` 这个结构体的内部顺序对于 `*nix` 和 `Windows` 稍有不同，以 `*nix` 为例

```
struct addrinfo{
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  socklen_t ai_addrlen;
  struct sockaddr * ai_addr; /* 存放结果地址的地
方 */
  char * ai_canonname; /* 忽略它吧，很长一段时间
你无须关注它 */
  struct addrinfo * ai_next; /* 一个域名/IP地址
可能解析出多个不同的 IP */
};
```

ii. `ai_family` 如果设定为 `AF_UNSPEC` 那么在调用 `getaddrinfo` 时，会自动帮你确定，传入的地址是什么类型的
iii. `ai_flags` 如果设定为 `AI_PASSIVE` 那么调用 `getaddrinfo` 且向其第一个参数传入 `NULL` 时会自动绑定自身 IP，相当于设定 `INADDR_ANY`
iv. `ai_socktype` 就是要创建的套接字类型，这个必须明确声明，系统没法预判(日后人工智能说不定呢?)
v. `ai_protocol` 一般情况下我们设置为 `0`，含义可以自行查找，例如 `MSDN` 或者 `UNP`
vi. `ai_addr` 这里保存着结果，可以通过 调用 `getaddrinfo` 之后 的第四个参数获得。
vii. `ai_addrlen` 同上
viii. `ai_next` 同上
ix. `getaddrinfo` 强大的接口函数

```
int getaddrinfo(const char * node, const char
 * service,
                     const struct addrinfo * hin
ts, struct addrinfo ** res);
```

x. 通俗的说这几个参数的作用
xi. `node` 便是待获取或者待绑定的 域名 或是 **IP**，也就是说，这里可以直接填写域名，由操作系统来转换成 **IP** 信息，或者直接填写**IP**亦可，是以字符串的形式
xii. `service` 便是端口号的意思，也是字符串形式
xiii. `hints` 通俗的来说就是告诉接口，我需要你反馈哪些信息给我(第四个参数)，并将这些信息填写到第四个参数里。
xiv. `res` 便是保存结果的地方，需要注意的是，这个结果在API内部是动态分配内存了，所以使用完之后需要调用另一个接口(`freeaddrinfo`)将其释放
xv. 实际上对于现代的 套接字编程 而言，多了几个新的存储 IP 信息的结构体，例如 `struct sockaddr_in6` 和 `struct sockaddr_storage` 等。

- 其中，前者是后者的大小上的子集，即一个 `struct storage` 一定能够装下一个 `struct sockaddr_in6`，具体(实际上根本看不到有意义的实现)

```
 struct sockaddr_in6{
   u_int16_t sin6_family;
   u_int16_t sin6_port;
   u_int32_t sin6_flowinfo; /* 暂时忽略它 */
   struct in6_addr sin6_addr; /* IPv6 的地址存放在此结构体中 */
   u_int32_t sin_scope_id;  /* 暂时忽略它 */
 };
 struct in6_addr{
   unsigned char s6_addr[16];
 }
 -----------------------------------------------------------
 struct sockaddr_storage{
   sa_family_t ss_family; /* 地址的种类 */
   char __ss_pad1[_SS_PAD1SIZE]; /* 从此处开始，不是实现者几乎是没办法理解 */
   int64_t __ss_align;           /* 从名字上可以看出大概是为了兼容两个不同 IP 类型而做出的妥协 */
   char __ss_pad2[_SS_PAD2SIZE]; /* 隐藏了实际内容，除了 IP 的种类以外，无法直接获取其他的任何信息。 */
   /* 在各个*nix 的具体实现中， 可能有不同的实现，例如 `__ss_pad1` ， `__ss_pad2` , 可能合并成一个 `pad` 。 */
 };
```

在实际中，我们往往不需要为不同的IP类型声明不同的存储类型，直接使用 `struct sockaddr_storage` 就可以，使用时直接强制转换类型即可

xvi. 改写上方 接收端 例子中，进入接收信息的状态部分

```
/* 首先将多于的变量化简 */
// - struct sockaddr_in host_v4; /* IPv4 地址 */
// - struct sockaddr_in6 host_v6; /* IPv6 地址
struct sockaddr_storage host_ver_any; /* + 任意类型的 IP 地址 */
...
/* 进入接收信息的状态部分 */
recvfrom(sock, mess, 15, 0, (struct sockaddr *)&host_ver_any, &addr_len); /* 像是又回到了只有 IPv4 的年代*/
```

xvii. 补充完整上方对应的 发送端 代码

```
int sock;
const char* mess = "Hello Server!";
char get_mess[GET_MAX]; /* 后续版本使用 */
struct sockaddr_storage recv_host; /* - struct sockaddr_in recv_host; */
struct addrinfo tmp, *result;
struct addrinfo *p;
socklen_t addr_len;

/* 获取对端的信息 */
memset(&tmp, 0, sizeof tmp);
tmp.ai_family = AF_UNSPEC;
tmp.ai_flags = AI_PASSIVE;
tmp.ai_socktype = SOCK_DGRAM;
getaddrinfo(argv[1], argv[2], &tmp, &result); /* argv[1] 代表对端的 IP地址， argv[2] 代表对端的 端口号 */

/* 创建套接字 */
for(p = result; p != NULL; p = p->ai_next)
{
  sock = socket(p->ai_family, p->ai_socktype, p->ai_protocol);  /* - sock = socket(PF_INET, SOCK_DGRAM, 0); */
```

```
    if(sock == -1)
      continue;
    /* 此处少了绑定 bind 函数，因为作为发送端不需要讲
对端的信息 绑定 到创建的套接字上。 */
    break; /* 找到就可以退出了，当然也有可能没找到，
那么此时 p 的值一定是 NULL */
  }
  if(p == NULL)
  {
    /* 错误处理 */
  }
  /* -// 设定对端信息
  memset(&recv_host, 0, sizeof(recv_host));
  recv_host.sin_family = AF_INET;
  recv_host.sin_addr.s_addr = inet_addr("127.0
.0.1");
  recv_host.sin_port = htons(6000);
  */

  /* 发送信息 */
  /* 在此处，发送端的IP地址和端口号等各类信息，随着这
个函数的调用，自动绑定在了套接字上 */
  sendto(sock, mess, strlen(mess), 0, p->ai_ad
dr, p->ai_addrlen);
  /* 完成，关闭 */
  freeaddrinfo(result); /* 实际上这个函数应该在使
用完 result 的地方就予以调用 */
  close(sock);
```

xviii. 到了此处，实际上是开了网络编程的一个初始，解除了现代的 UDP 最简单的用法(甚至还算不上完整的使用)，但是确实是进行了交互。

# #

- 首先介绍 UDP 并不是因为它简单，而是因为他简洁，也不是因为它不重要，相反他其实很强大。
- 永远不要小看一个简洁的东西，就像 C语言

- 下一篇将详细记录 **UDP** 的相关记录

在这之前

- 首先还是科普记录一下协议的知识。
- 阮一峰的博客 ： 互联网协议入门(一)
- 阮一峰的比克：互联网协议入门(二)
- 上述两篇文章十分浅显易懂，十分符合科普二字，下方将对上述两个文章进行适当的补充。

ARP 协议

- 最简便的方法就是找一个有 WireShark 软件或者 `tcpdump` 的 `*nix` 平台，前者你可以选择随意监听一个机器，不多时就能看见 **ARP** 协议的使用，因为它使用的太频繁了。
- 对于 ARP 协议而言，首先对于一台机器 A，想与 机器B 通信，(假设此时 机器 A 的高速缓存区(操作系统一定时间更新一次)中 没有 机器B的缓存)，
    - 那么机器A就向广播地址发出 **ARP**请求，如果 机器B 收到了这个请求，就将自己的信息(IP地址，MAC地址)填入 **ARP**应答 中，再发送回去就行。
    - 上述中， **ARP**请求 和 **ARP**应答 是一种报文形式的信息，是 ARP协议 所附带的实现产品，也是用于两台主机之间进行通信。
    - 这是当 机器A 和 机器B 同处于一个网络的情况下，可以借由本网络段的广播地址 发送请求报文。
- 对于不同网络段的 机器A 与 机器B 而言，想要通过 **ARP**协议 获取 **MAC**地址，就需要借助路由器的帮助了，可以想象一下，路由器(可以不止一个)在中间，机器A 和 机器B 分别在这些路由器的两边(即在不同子网)
    - 由于 A 和 B 不在同一个子网内，所以没办法通过通过直接通过广播到达，但是有了路由器，就能进行 **ARP**代理 的操作，大概就是将路由器当成机器B， A向自己的本地路由器发送 **ARP**请求
    - 之后路由器判断出是发送给B的ARP请求，又正好 B 在自己的管辖范围之内，就把自己的硬件地址 写入 **ARP**应答 中发回去，之后再有A向B 的数据，就都是A先发送给路由器，再经由路由器发往B了
    - 一篇比较好的资源是 Proxy ARP

ICMP

- 这个协议比较重要，后方的概念也会涉及。

- 请求应答报文 和 差错报文 ，重点在于差错报文。
- 请求应答报文在 `ICMP` 的应用中可以拿来查询本机的子网掩码之类的信息，大致通过向本子网内的所有主机发送该请求报文(包括自己，实际上就是广播)，后接收应答，得到信息
- 差错报文在后续中会有提到，这里需要科普一二。
- 首先对于差错报文的一大部分是关于 **xxx**不可达 的类型，例如主机不可达，端口不可达等等，每次出现错误的时候，ICMP报文总是第一时间返回给对端，(它一次只会出现一份，否则会造成网络风暴)，但是对端是否能够接收到，就不是发送端的问题了。
- 这点上 套接字的类型 有着一定的联系，例如 UDP 在 `unconnected` 状态下是会忽略 ICMP报文的。而 TCP 因为总是 `connected` 的，所以对于 ICMP报文能很好的捕捉。
- ICMP差错报文中总是带着 出错数据报中的一部分真实数据，用于配对。

> 注意，对于 `UDP` 而言，只有 `connected` 状态下，才会收到 **ICMP** 报文，可以通过 `errno == ECONNREFUSED` 来确定，具体来说就是在你发送完本次数据之后，的下一次系统调用时会有这个想想，代表你的小心没有被送到对方手里，而是被对方丢弃了。
>
> 在没有一个完备的思路以及良好的设计之前, `UDP` 是一个十分艰难的挑战，只有在 `TCP` 实在无法满足我们的性能需求时，我们才来重新考虑 `UDP` 。

0x13-套接字编程-HTTP服务器(1)

> 这里不是百科全书，所以只会用最简单，最明了的语言，来讲最实用的TCP编程。
>
> `Echo` 程序太多，就不再重复了，贯穿整个章节的将会是一个 **HTTP** 服务器
>
> 这回是在 `Linux` 下开发，而不是 `Windows`

TCP

- 囫囵吞枣般的喂完**UDP**的基本应用，以及一些**API**的使用，不再赘述**TCP**的使用，其实是很多我也不懂，(逃。
- 但是，对于**API**我一向抱以用多少学多少知多少的态度，人生如戏，重在看戏啊。
- 如果想要查找具体的完整的API可以先去查，**UNIX**网络编程：卷**1** + **Linux Man**手册，其中前者有一些部分实际上已经过时(内核版本跟不上)。更不用说后续加入**Linux**的一些接口，例如 `epoll` 。但是其他的接口还是可以参考的，并且十分的详细。
- TCP，这是一个极其复杂的协议，说复杂是因为在这几十年的发展中，对其的优化已经多到令人发指，于此而言，虽说即便不知道这些优化也是可以编写程序，但是建议还是能够熟悉一下流程(两端交互的过程)
    - 三次握手，四次挥手，拥塞控制，滑动窗口机制。
    - 对于前两个而言，我个人有不同见解，于为什么是三次握手，而不是其他次数，但是由于并不一定被接受，所以我不在这里写入，有兴趣的可以**Emali**我一起讨论，在面试的时候，我也是会和面试官讨论这个问题，但一直没有满意的答案。

HTTP

- 说了贯穿本章节的是一个 **HTTP**服务器，如此就一定要说说 **HTTP**协议 了，如果说前面的**TCP**即使你不懂它的原理也能编写一个能运行且效率不错的**TCP**程序的话，那么想写一个**HTTP**服务器，你要是不懂 **HTTP**协议，简直是寸步难行。
- 最权威的莫过于 《**HTTP**权威指南》，你可别想着去看HTTP的标准草案了，那真是神鬼难懂。当然这本书亦是一部大块头，可以选择在网上找一些HTTP协议的资料来补充一些基本知识，再用这本大块头来检索自己需要了解的地

方。

- 简单说一下 **HTTP** 协议
    - 实际上这个协议就是 TCP协议在 应用层 的一种封装，换句话说 **HTTP**服务器实质上还是一个 **TCP** 程序，只不过 **TCP**协议 已经被操作系统实现好，留出借口来给你调用，而 **HTTP**协议 则是完全需要你自己编写。
    - 所谓协议就是双方都需要遵守的一个规则，所以 **HTTP**协议 实质上是一种非实际的东西，最主要的还是包括了一个 状态机，称为 **HTTP**状态机 ，它的作用就是 解析/生成 **HTTP**报文，不必太过注意这些名词，最开始想写这个程序的时候，也没有太多顾虑这种名词性的东西。
    - 所有的名词都是为了更好的抽象一件事物，就好比数学上的各种 符号，其实本没必要存在，但为了抽象简单的表达，符号就应运而生。所以当你理解不了一个符号的时候，就跳过他，因为你总能找到一个它的替代物(例如符号对应的 公式/展开式，符号对应的定理，定理对应的实际例子)，而且比他更加的详细且好理解，这是我三年学习数学总结出来的经验。
    - 扯了一些没用的，回归正题
- 协议版本号
    - 主流的就是 **HTTP/1.0**, **HTTP/1.1**，其他的在编写本程序时不必太在意
    - 需要了解的是这两个版本好的一些功能区别，例如最广为人知的 `Connection:` 的默认属性
- 方法
    - `GET`
    - `POST`
    - `HEAD`
    - 实际上，在一般的服务器实现中，也只需要实现这三个方法就够了，特别是前两者比较重要，将以 `GET` 进行程序的方法编写，有兴趣的可以自己实现 `POST` 方法，我的源代码中也已经完成一半了，但并不打算继续完成。

> 有图有真相

```
GET / HTTP/1.1\r\n              <--- 这是状态行，包括一个请求方法，资源，协议版本
Host: www.wushxin.top\r\n       <-- 这是属性头，
Connection: keep-alive\r\n      <-- ...
\r\n                            <-- 一直到空行结束
```

- 这是最简单的HTTP请求报文，整个报文的意思是请求 资源根目录(注意，是资源根目录) 下的资源(默认 请求根目录且不写什么资源就返回 `index.html`)，并希望与服务器保持连接。

> 这里的保持连接和 TCP 协议中的保持连接不一样，具体可以去查找资料，简单来说这是一种不可靠的保持连接，双方都可以在做出保证之后，突然断开连接。可以把它当成不靠谱连接，但是表面工作还是要做的。
>
> 有图有真相

```
200 OK\r\n                    <-- 状态行，包括一个状态码，状态详情
Host: www.wushxin.top\r\n     <-- 属性，
Connection: close\r\n         <-- ...
Content-Length: 78\r\n        <-- 这个属性，代表空行后面数据有多少
\r\n                          <-- 直到空行
<html><body><p>Hello, That is the Resource which you Request!</b
ody></html>
```

- 这是很简单的一种返回报文， `Content-Length` 属性在这里特别重要！绝对不能缺少，它给对端一种信息就是，这个报文的结束位置在哪里。
- 在上述报文中，每行的末尾都有 `\r\n` 这是**HTTP**协议 用来表达行末的一个标志，而且，**HTTP**协议 的头部和文本部有一个空行 `\r\n` 来进行分割。
- 以上就是所有我想要介绍的 **HTTP**协议的内容，有些零散，还是建议去查找一些资料后再来往下看。

详细计划

- 上面的内容看起来有些零碎，但是做一个程序，还是需要理一理自己的思路。
- 首先，我们的目的是做出一个 并发**HTTP**服务器， 会涉及到的知识点：
  1. **HTTP**协议的实现
  2. **TCP**的Socket编程(即套接字编程)，所谓网络编程
  3. 多线程
  4. `epoll` 机制，暂时不要管 `select` 和 `poll` 这两个相似功能的机制
  5.
     - FastCGI的实现(作为扩展，有兴趣的在最后可以去尝试，资源:FastCGI规范)

0x14-套接字编程-HTTP服务器(2)

HTTP服务器的结构

- **HTTP**服务器 本质上就是一个 **TCP**的接收端 程序
- 但凡一个正常的 **TCP** 接收端程序，都逃不过那几个流程：
    - 创建监听 `socket` -> 绑定端口，**IP** -> 监听 `socket` -> 接受新连接 -> 处理读写**...** -> 关闭完成的连接
    - 其中前三步比较固定，最多对这个监听用的 `socket` ，进行一些优化处理，设置一些属性之类的，但那都是固定模式，想想就能明白。硬要说重要的地方，也就是在于是否把 `socket` 设为非阻塞(**non-blocking**)了。
    - 后面几步，每个都是很重要的环节，需要细细设计才行

> 所谓非阻塞，我还是不班门弄斧了，请移步 **UNIX**网络编程**-**卷**1-**中文**·**第三版 **127**页**(**英文版**160**页**)** 的图 `6-6` ，清楚的对比了，阻塞，非阻塞，异步，I/O复用的区别和含义。十分建议写网络程序之前，去把这本书的某些章节大致过一遍。

- 对于这章节需要写的这个服务器而言，采用的是经典且流行的 **I/O**复用**+**非阻塞套接字**(socket)+**多线程**(**线程池**)** 结构。
- 呐，又出现一个新的知识点，**I/O**复用，这是什么鬼。

I/O复用

- 我给一个不太严密的解释，那就是 将你这个程序需要等待的地方，集中起来
- 打个比方：
    - 假设有100个新连接，被你的监听套接字给成功接受(`accept()`)了
    - 这时候并不是所有新连接都立刻有数据可以读，那此时你有两种选择：阻塞，非阻塞。但不论是哪一种都会导致同一个结果
        - 阻塞： 那假设你只有一个线程处理这100个连接，万一要是正好处理到这个暂时没有数据的连接，就要一直等待它的数据到来，后面的几十个连接都要闲着；假设有多个线程同时处理，理由还是一样，换汤不换药，而且难道你还能开100个线程去处理吗？那如果更多的连接呢？
        - 非阻塞： 比阻塞看起来稍微好一些，因为如果没有数据到来的话，那就直接跳过这个连接，直接去处理下一个连接了，但是你想想，这不就是遍历了吗？万一连接量一大，假设上万，而且只有少数的几个连

    接有数据活跃，这无用功做的是不是太多了？多开几个线程去平摊压力？那么要开多少比较合适？
  - 这时候喜欢偷懒的程序员，自然就不愿意了，于是考虑是否可以有一个，让我们可以在单个线程的情况下还能够只处理那些活跃的连接？
  - 这时候出现了所谓的 **I/O**复用 技术，说是技术，因为它使用的还是同步型的操作(`read, write`)，只不过套接字设为非阻塞的了。
  - Linux平台下的 `epoll` , Unix(包括Mac)平台下的 `kqueue` , Windows平台下的 `IOCP` ，各平台通用的 `select` , `poll` ，还有几个历史实现就不赘述了。
  - 最后这两个 `select` , `poll` 在活跃连接明显少于总连接数的情况下，性能比前三个要差许多，故本章使用的是 `epoll` ,(当然还有资料比较多的原因啦
- 说说 `epoll` 的工作
  - 首先它帮我们管理着所有的套接字，用来监听这些套接字哪些有了数据，就返回谁。
  - 将所有等待，阻塞都集中在了一个地方，那就是 `epoll_wait()` 调用上
  - 而且可以针对不同的事件进行不同的监听，这就是事件驱动这种模式的由来
- 事件驱动
  - 简单来说，就是针对某种事件进行触发的一种编程模式
  - 具体来说，假设你在网络编程，正在处理一个套接字，由于TCP是全双工的，意味着这个TCP套接字是可读可写，问题来了，什么时候可读，什么时候可写呢？这就延伸出了事件，读事件，写事件，错误事件等
  - 可以通过 `epoll_clt()` 来设置要监听的事件，当然也可以同时监听多个事件，看你的设计了。
- 具体的 `epoll` 接口的详细介绍，可以直接在**Linux**上，使用 `man epoll` 进行查看手册，这是基本功。
  - `epoll_create` , `epoll_ctl` , `epoll_wait`

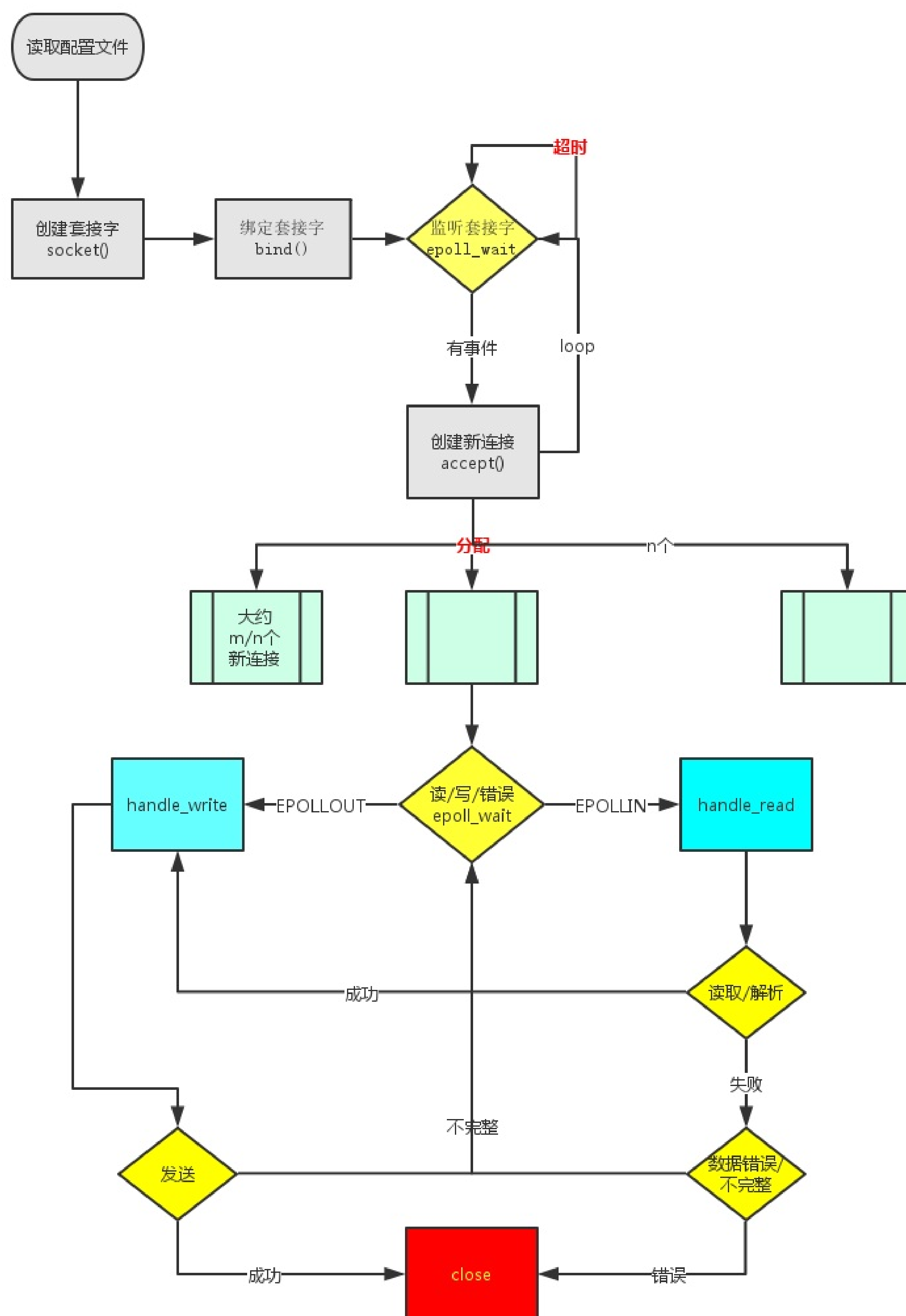
服务器结构

- 继续回到服务器结构
- 上面简单的讲述了一下什么是 **I/O**复用，以及将会用到的具体实现 `epoll` 。那具体说一下，整个程序的流程
- 还是老规矩，写程序之前要先构思，自己在纸上画一画，大概的流程是什么

- 问题： 想要完整处理一个HTTP请求，需要哪些步骤？
  1. 解析**HTTP**请求报文
  2. 创建**HTTP**回复报文
- 逻辑就这么简单啊，但是加上细节部分，就会稍微麻烦一些了：
  - 完整地 从套接字中，读取 **HTTP**请求报文
  - 解析 **HTTP**请求报文，并判断其有效性
  - 生成 **HTTP**回复报文
  - 完整地 通过对应套接字，发送给请求者。
- 在这里我假设，你已经对**TCP**编程的模型很熟悉了，不熟的可以去顶部看看再回来
  - 并发服务器的关键点就在于
    - 高效且正确地接收尽可能多的连接
    - 高效且正确地处理尽可能多的连接
    - 以上忽略了安全性
  - 该如何设计？
    - 让某个 `epoll` 用来服务于接收新连接这个环节(`accept`)
    - 让某些 `epoll` 用来处理这些新连接的事务。
    - 这样理论上我们既发挥了单核的极限(epoll)，又用上了多核的优势(多个 `epoll`)
  - 更具体的呢？
    - 在主线程里使用单个 `epoll` 来处理，监听套接字的读事件，也就是接受新连接
    - 再开几个线程 `epoll` ，用来平分处理这些新连接。
  - 这样也就是网络编程的一整个流程，如果看到这里你已经大概有了一个程序思路，实际上就已经达到目的了，接下来就是直接上手代码就行
  - 还是迷迷糊糊的，就一步一步跟着我，写出这个服务器，会大有脾益。

> 小经验，在编程中，读往往比写要复杂许多。在网络编程里面亦是。

- 有图有真相，希望能够自己画。



- 现在大致有了思路，可以整理整理自己接下来该干什么了

环境准备

- 99%的中国大学学生的操作系统，应该都是 Windows或者Max OS(maxOS)，那么建议你直接使用虚拟机进行环境的搭建，可以选择开源免费的 `Visual Box` ，Windows下也可以使用商业版的 `VMware` ，Mac下有一个更棒的商业版选择 `Paralelle Desktop` ，但是这都是软件，算是无关紧要的。
- 选择一个 `Linux` 发行版，由于我用的是 `Debian` 系列的 `Ubuntu 16.04 LTS` ，所以我也推荐这个发行版，其他的发行版也许略有差异，不再多说。
- 装好之后，直接进入开发阶段吧。
    - IDE可以选择 `Clion` 或者 `Kdevelop` 。
    - 当然你要用 `Vim` 我也不会阻拦，但是请装好两个插件 `Nerdtree` 和 `YouCompleteMe` ，配合好另一个软件 `tmux` (简单使用)，不然你会想死。
    - 除了 `Vim` ，你也可以选择 `Visual Studio Code` 加装一个 `C/C++ tools` 也是不错的。
    - 作为时尚的我，自然选择 `Clion` 了，简单明了，且还是使用 `CMake` 作为构建工具。
- 想要进行这么底层的网络编程，请准备好 `Google` 和 `Unix网络编程卷1` ，如果你两个都没有的话，不说了，再见。建议准备一个那玩意儿去访问 `Google` 。

0x15-套接字编程-HTTP服务器(3)

- 在一切开始之前，我们需要设想一下，为了让自己的HTTP服务器变得更加灵活，我们可以让某些参数不必硬编码进程序中，而是用配置文件的方式读取
- 一个HTTP服务器的基本配置无非是
    - IP地址，端口号， 根目录路径
    - 额外增加一个 线程数
    - 实际上，应该不需要我们人为指定，但为了调试方便，所以选择放在配置文件中
- 接下来我们写一个可以解析配置文件的小模块函数

```
struct init_config_from_file {
    int  core_num;              /* CPU Core numbers */
#define PORT_SIZE 10
    char listen_port[PORT_SIZE]; /*  */
#define ADDR_SIZE IPV6_LENGTH_CHAR
    char use_addr[ADDR_SIZE];    /* NULL For Auto select(B
y Operating System) */
#define PATH_LENGTH 256
    char root_path[PATH_LENGTH]; /* page root path */
};
typedef struct init_config_from_file wsx_config_t;
```

这个是配置文件的所有属性，可以将读取的参数，存进这个结构体中，与主线程交互

```
/*
* Read the config file "wsx.conf" in particular path
* and Put the data to the config object
* @param  config is aims to be a parameter set
* @return 0 means Success
* */
int init_config(wsx_config_t * config);
```

交互的接口，我的配置文件叫做 wsx.conf

> 对于配置文件存放位置而言，可以灵活一些，例如可以额外添加一个命令行参数，用来指定本次需要使用的配置文件路径： `./httpd -f /path/to/wsx.conf` 当然这用在开发版本可以方便调试，实际上的HTTP服务器并不行，参见守护进程的定义
>
> 最经典的做法还是指定默认路径，将配置文件都存放在某个地方，可以多设定几个，并设定优先级

- 想想，我们需要什么功能，我给自己的配置文件添加了注释功能，以 `#` 开头的都是注释，这点十分容易做到。
- 上代码

```
  static const char * config_path_search[] = {CONFIG_FILE_PATH, "./wsx.conf", "/etc/wushxin/wsx.conf", NULL};

  int init_config(wsx_config_t * config){
      const char ** roll = config_path_search;
      FILE * file;
      for (int i = 0; roll[i] != NULL; ++i) {
          file = fopen(roll[i], "r");
          if (file != NULL)
              break;
      }
      if (NULL == file) {
  #if defined(WSX_DEBUG)
          fprintf(stderr, "Check For the Config file, does it stay its life?\n"
          "In Such Path: \n%s\n%s\n%s\n", config_path_search[0], config_path_search[1], config_path_search[2]);
  #endif
          exit(-1);
      }
  ...未结束
```

这是很简单的文件操作，包括打开文件，验证是否成功，可以选择将其封装成一个 `inline` 函数，来模块化这个逻辑。

```
    char buf[PATH_LENGTH] = {"\0"};
    char * ret;
    ret = fgets(buf, PATH_LENGTH, file);
    while (ret != NULL) {
        char * pos = strchr(buf, ':');
        char * check = strchr(buf, '#'); /* Start with # w
ill be ignore */
        if (check != NULL)
            *check = '\0';

        if (pos != NULL) {
            *pos++ = '\0';
            if (0 == strncasecmp(buf, "thread", 6)) {
                sscanf(pos, "%d", &config->core_num);
            }
            else if (0 == strncasecmp(buf, "root", 4)) {
                sscanf(pos, "%s", &config->root_path);
                /* End up without "/", Add it */
                if ((config->root_path)[strlen(config->roo
t_path)-1] != '/') {
                    strncat(config->root_path, "/", 1);
                }
            }
            else if (0 == strncasecmp(buf, "port", 4)) {
                sscanf(pos, "%s", &config->listen_port);
            }
            else if (0 == strncasecmp(buf, "addr", 4)) {
                sscanf(pos, "%s", &config->use_addr);
            }
        } /* if pos != NULL */
    ret = fgets(buf, PATH_LENGTH, file);
    } /* while */
    fclose(file);
    return 0;
}
```

真正的核心代码没几行，四个 `if` ,使用 `strncasecmp` 函数，检测参数。但是并没有 验证参数的正确性。

当然你也可以写成 `json` 的形式，再用第三方库，比如`c-json`之类的解析，但 那不是要依赖第三方了吗？所以我的建议还是自己写一个解析的函数。

如果没能理解这小段代码，建议翻一下C语言的入门教材，回顾一下语法。

- 配置文件的样式

```
  # Just Edit this Config file Or
  # You can Create a new one and save the Old to
  # Back up
  # But Remember that , that file can only parse
  # the FOUR CONFIGURATION :
  # thread root port address
  # Watch out the case sensitive !!!
  # thread -- For the Worker thread number
  # root   -- For the WebSite's root path
  # port   -- Listen Port
  # address -- Host's address(Note it If you can)
  #            Or empty For the auto select by Operating Sys
tem
  thread:8
  # Using shell Command (pwd) to show your root Path!
  root:/root/ClionProjects/httpd3/
  port:9998 # That is a port
  address:192.168.141.149
```

- 配置文件读取完成了，我们是时候设计一下主函数的流程了，回想一下流程图，下一步就应该创建套接字，绑定，并监听(**`listen`**)了！（流程图中没有画出 `listen` ，过于冗余，但却必不可少）
- 可以将 创建，绑定合并成一个函数，在成功之后，再执行 `listen` 。

```
  /*
    * Open The Listen Socket With the specific host(IP addres
s) and port
    * That must be compatible with the IPv6 And IPv4
    * host_addr could be NULL
    * port MUST NOT BE NULL !!!
    * sock_type is the pointer to a memory ,which comes from
the Outside(The Caller)
    * */
  int open_listenfd(const char * restrict host_addr,const ch
ar * restrict port, int * restrict sock_type);
```

可以看出来，需要一个IP, 一个PORT， 第三个参数是套接字类型担不是传入参数，而是传出参数。

```
  int open_listenfd(const char * restrict host_addr, const c
har *     restrict port, int * restrict sock_type){
      int listenfd = 0; /* listen the Port, To accept the ne
w Connection */
      struct addrinfo info_of_host;
      struct addrinfo * result;
      struct addrinfo * p;

      /* 实际上这一行完全可以在上面使用 初始化来达到目的。
       * struct addrinfo info_of_host = {0}; 需要c99
       */
      memset(&info_of_host, 0, sizeof(info_of_host));
      info_of_host.ai_family   = AF_UNSPEC; /* Unknown Socke
t Type */
      info_of_host.ai_flags    = AI_PASSIVE; /* Let the Prog
ram to help us fill the Message we need */
      info_of_host.ai_socktype = SOCK_STREAM; /* TCP */

      int error_code;
      if(0 != (error_code = getaddrinfo(host_addr, port,
 &info_of_host, &result))){
          fputs(gai_strerror(error_code), stderr);
          return ERR_GETADDRINFO; /* -2 */
      }
```

```
      for(p = result; p != NULL; p = p->ai_next) {
          listenfd = socket(p->ai_family, p->ai_socktype, p-
>ai_protocol);

          if(-1 == listenfd)
              continue; /* Try the Next Possibility */
          optimizes(listenfd);
          if(-1 == bind(listenfd, p->ai_addr, p->ai_addrlen)
){
              close(listenfd);
              continue; /* Same Reason */
          }
          break; /* If we get here, it means that we have su
cceed     to do all the Work */
      }
      freeaddrinfo(result);
      if (NULL == p) {
          fprintf(stderr, "In %s, Line: %d\nError Occur whil
e Open/    Binding the listen fd\n",__FILE__, __LINE__);
          return ERR_BINDIND;
      }
      fprintf(stderr, "DEBUG MESG: Now We(%d) are in : %s ,
listen     the %s port Success\n", listenfd,
      inet_ntoa(((struct sockaddr_in *)p->ai_addr)->sin_addr
), port);
      *sock_type = p->ai_family;
      set_nonblock(listenfd);
      return listenfd;
  }
```

其中有一个optimizes,是用来设置一些套接字选项的，现在只需要知道有这些选项就行

套接字选项分别是 `TCP_NODELAY` 和 `SO_REUSEADDR` 。

细看之下，和前面介绍的几个接口几乎是完全一致的用法。但如果认为网络编程就是这样接口调用的话，那就是大错特错。

就这样，如果你的配置文件中，没什么差错的话，我们就完成了打开服务器套接字的工作，这时候你可以组织并且运行一下前面说的这些代码，看看是否如此。

> 运行成功与否可以通过你的终端是否显示上述的调试信息看出来:
>
> **DEBUG MESG: Now We(x) are in : %s , listen the xx port Success**

- 写到这里，实际上整个主函数的代码已经接近尾声，来看看全部的过程调用

```
int main(int argc, char * argv[]) {
    wsx_config_t config = {0};
    init_config(&config)

    int sock_type = 0;
    int listenfd = open_listenfd(config.use_addr, config.l
isten_port, &sock_type);
    listen(listenfd, SOMAXCONN);
    signal(SIGPIPE, SIG_IGN);
    handle_loop(listenfd, sock_type, &config);
    return 0;
}
```

这个逻辑已经十分清晰，为了方便我省去了错误检查，在代码中应该自己添加，这里面有两个新事物： `signal()` ， `handle_loop()`

- 来解释一下 `signal(SIGPIPE, SIG_IGN)` 是什么以及为什么
  - `signal` 是信号函数，还记得之前的章节用它来当做函数指针类型的一个练习思考题吗？它的作用就是在本进程/线程接收到该信号(`SIGPIPE`)时候,会进行这样的(`SIG_IGN`)处理
  - 当然它有更好更推荐的做法 `sigation` ,比较复杂但是也比较推荐你用它，这里为了减少概念，就用了最原始的 `signal` 。
  - `SIGPIPE` 是一个关于写的错误，触发条件是向一个发送了 `RST` 的对端进行写操作，默认行为就是结束本进程，我们当然不愿意结束了，明明是对方的错，怎么要我们死。最基本的做法就是忽略它 `SIG_IGN` 。
  - 稍微解释一下 `SIGPIPE` ，模拟一下情形，这里需要对TCP的工作方式有一定了解，不了解的可以跳过：
    - TCP是全双工的，意味着可读可写，假设有A,B端，本来工作的好好

        的，突然B端崩溃退出了，那自然联系A,B端的套接字连接就断了，但是A端并不懂啊，它这时候只知道B端不会再发送消息给自己了(因为接到了B发给自己的FIN，自己回复了ACK，关闭了接收通道)，并不懂自己还能不能发消息给B啊(所以A当做自己能发给B端)
        - 然而实际上，现在哪里还能发消息给B啊，这就回到了上面，如果向一个发送了 `RST` 的对端进行写操作的话，就会触发 `SIGPIPE` ,信号这个东西就是全局的，所以如果你想知道哪个线程触发了这个信号，还需要检查写操作是否返回了 `EPIPE` 错误
    - 看不懂也无所谓，来日方长，细水长流。这就是这一行代码的意义，就是为了忽略这个信号。
- `handle_loop` 是一个事件循环的入口
    - 就是所有的事务处理准备都在里面，回想一下流程图，我们接下来该干什么
    - 使用 `epoll` 监听服务器套接字，用来建立新连接
    - 分配新连接给子线程，在其中处理各种事件。
    - 呐，实际上 `handle_loop` 就干了两件事
        - 准备一下服务器资源(包括存储新连接的各种信息)
        - 创建子线程用来 监听服务器套接字 或 处理新连接事件
- 几个全局变量

```
  static int * epfd_group = NULL;   /* Workers' epfd set */
  static int   epfd_group_size = 0; /* Workers' epfd set size */
  static int   workers = 0;         /* Number of Workers */
  static int   listeners = MAX_LISTEN_EPFD_SIZE; /* Number of Listenner */
  static conn_client * clients;   /* Client set */
```

- `handle_loop()`

```
  void handle_loop(int file_dsption, int sock_type, const ws
x_config_t * config) {
      workers = config->core_num - listeners;
      int listen_epfd = epoll_create1(0);
      { /* Register listen fd to the listen_epfd */
          struct epoll_event event;
          event.data.fd = file_dsption;
          event.events = EPOLLET | EPOLLERR | EPOLLIN;
          /* 以ET方式监听file_dsption的读事件，错误事件 */
          epoll_ctl(listen_epfd, EPOLL_CTL_ADD, file_dsption
, &event);
      }
      /* Prepare Workers Sources */
      prepare_workers(config);
      pthread_t listener_set[listeners];
      pthread_t worker_set[workers];
      for (int i = 0; i < listeners; ++i)
          pthread_create(&listener_set[i], NULL, listen_thre
ad, (void*)listen_epfd);
      for (int j = 0; j < workers; ++j) {
          pthread_create(&worker_set[j], NULL, workers_threa
d, (void*)(epfd_group[j]));
          pthread_detach(worker_set[j]);
      }
      for (int k = 0; k < listeners; ++k)
          pthread_join(listener_set[k], NULL);
      destroy_resouce();
  }
```

使用了最原始的线性数组来存储所有的连接信息（ `conn_client` ），这其实弊端很大，比如最明显的数量以及预分配的资源过大。但关键是够简单，且效率最高。

整个的原理就是，在接受到新连接以后，按照某种规则分配给第i个子线程，每个子线程中有一个工作 `epoll` (epoll_group[i-1])，用来监听新连接的事件，并处理。

`prepare_workers` 就是分配内存空间的相关工作。这段代码，同样省略了错误检查，希望自己添加。

`{}` 里面可以看出来怎么向 `epoll` 实例中注册监听实体，以及监听事件。

整段代码的后半部分，是关于线程的启动，操作，销毁。 `pthread_detach` 意味着放弃线程的资源回收权，用通俗的话来说就是："撒丫子跑吧，我管不着你了！"。

- 这就是完整的一个主函数逻辑，实际上非常简单，到现在为止也没出现过十分复杂的东西，就像在做繁琐的准备工作一样。

下一节将会详细讲解

1. 连接信息都有哪些需要存储的
2. 如何处理读事件，字符数据的管理呢？

0x16-套接字编程-HTTP服务器(4)

新连接

1. 一个新晋连接，有哪些信息是值得我们关注的？
2. 该如何存储它们？

> 这里将会叙述的并不会很完整，因为不同目的的网络程序，需要关注的信息也大不相同
>
> 特别是这个程序关注的是如何使用C语言编写一个服务器

1. 我们最关心的，还是对端通过这个新连接所发来的信息
    - 简单来说就是我们 `read` 到的信息。进行过系统编程的都应该会知道这个函数,与之对应的是 `write` 。与 **C**标准库 为我们提供的标准格式化输入输出不同的地方在于其操作的对象。 `read/write` 操作的是一个在叫做 文件描述符**(file description)** 的 `int` 类型的东西，而标准库的函数(`printf/scanf`)操作的则是一个 `FILE*` 特殊的结构体指针，这两者之间可以互相转换，通过 `fdopen(fd-->FILE*)/fileno(FILE*-->fd)` 具体相关知识，查阅相关信息，如著名的 `APUE` 。
2. 其次我们对这个信息做相应处理，中间会有很多状态，也就是常常听到的**HTTP**状态机
    - 实际上也就是几个状态值在转换和过渡，只是名字专业了一些
3. 最后我们会生成一个信息，用来回复对端
    i. 这个也叫做响应报文

> `*nix` 下的文件描述符**(file description)**在 `Windows` 下近似相当于 文件句柄**(file handler)**，只不过前者是有规律的递增，而后者则不是。

1. 如何存储？

```
typedef unsigned char boolean;
struct connection {
    int  file_dsp;
#define CONN_BUF_SIZE 512
    int r_buf_offset;
    int w_buf_offset;
    string_t r_buf;
    string_t w_buf;
    struct {
        /* Is it Keep-alive in Application Layer */
        boolean conn_linger  : 1;
        boolean set_ep_out   : 1;
        boolean is_read_done : 1; /* Read from Peer Done? *
/
        boolean request_http_v : 2; /* HTTP/1.1 1.0 0.9 2.0
 */
        boolean request_method : 2; /* GET HEAD POST */
        int content_type : 4;  /* 2 ^ 4 -> 16 Types */
        int content_length; /* For POST */
        string_t requ_res_path; /* / */
    }conn_res;
};
typedef struct connection conn_client;
```

其中有一个陌生的事物，`string_t`，这个是用来进行字符串操作的一个自己写的结构，用于简化操作，可以把它看成一个可以自动增长的字符串类型。

再者就是，内嵌结构体中使用到了 `位域` 这个方式，主要是因为C中没有原生的 `bool` 类型，使用 `int` 来表示又太过奢侈

这个位域的写法在某些人看来似乎不太感冒，实际上还有替代的方法可以用，也就是使用掩码的思想，在一个 `int` 型中的不同位包含不同的信息，实际上和我这个的原理是相同的，只不过我将它拆开了，这样就可以不写各种处理宏

```
/* 另一种写法 */
...
struct {
    int status_set;
    int content_length;
    string_t request_length;
}conn_res
...
enum {
    SET_CONN_LINGGER = 1,
    SET_EPOLLOUT = 1 << 1,
    ...
}
/* 几乎对于每一个位置的操作都有三个，设置，复位，检测 */
#define SET_CONN_LINGER(MASK_SET) (MASK_SET &= SET_CONN_LIN
GER)
#define CLR_CONN_LINGER(MASK_SET) (MASK_SET &= (~SET_CONN_L
INGER)&0xFFFF)
#define IS_CONN_LINGER(MASK_SET) (MASK_SET & SET_CONN_LINGE
R)
```

依此类推。

> 实际上，对于这个 `string_t` 的设计是一个想当然的失败，当时是想尝试使用面向对象的想法，但是没有考虑到其使用时候的冗余，后面会看到这个小麻烦，但是总体上还是可以得。
>
> 这次总结出来的就是，在C里面使用面向对象的思维实在有点勉强，具体等后方说到这个 `string_t` 时会再提到。
>
> 2016-08-28 将其修改为正常的C风格。

1. 所以实际上来看一看，我存储了哪些状态信息
    - 一个新连接的 **file description** `file_dsp` ：这个肯定是必要的，不然你怎么对这个新连接进行操作。
    - 一个读缓冲配着一个读位移(`r_buf` 和 `r_buf_offset`)：
        - 之所以需要位移，是因为你要牢牢记住，尤其是在网络通信中，总会出现网络不稳的状况，这会导致某时候你的信息不能完全一次新的读

取到，也就是需要分次读取，所以你需要知道上次你读到哪里
        - 另一个原因是因为，在解析读取的信息的时候，你要时刻知道自己处理到哪里了，是否接收到数据不完整？是否接收的数据有错？等等。
    - 一个写缓冲配着一个写位移('w_buf'和 `w_buf_offset`)
        - 写事件要比读事件简单许多。
    - 一个包含**HTTP**状态的属性结构 `conn_res`
        - `conn_linger` : 是否保持连接(keep-alive)
        - `set_ep_out` : 是否设置监听写事件(EPOLLOUT)
        - `is_read_done` : 是否已经读取信息完毕
        - `request_http_v` : **HTTP**协议版本
        - `request_method` : **HTTP**请求方法
        - `content_type` : 响应报文 中的 属性
        - `content_length` : 同上
        - `requ_res_path` : 对端想请求的资源
2. 所以这也从另一个方面回答了上面的第二个问题 该如何存储它们？

3. 了解过，要存储那些信息，该如何存储这些信息之后，就能继续服务器的编写

事件循环

- 前面我们的进度，已经到了 `handle_loop` 里面，并且将总体流程已经过了一遍
- `handle_loop` 就是一个事件循环，我们整个程序的编程模型就是一个 事件驱动 的编程体系，什么是事件驱动，可以查阅相关资料，如 **UNP** 等书籍。在这个事件循环中，我们使用两个事件驱动我们的流程：读事件， 写事件
- 即，一旦某个连接可读（回忆一下**TCP**连接可读可写）我就处理读事件，写事件也是如此。
- 在这个循环中，我们启动了两种线程，一种专门用于接受建立新连接，一种专门用来处理新连接的读写事件，分别是 `listen_thread` 和 `workers_thread` ，常理来说前者一个就够了，后者可以酌情处理。
- 先说说比较简单的 `listen_thread`

listen_thread

- 回到 `handle_loop` 的代码中可以看到有一个独立的代码块 `{}` ，这个代码块的作用就是将我们之前创建的服务器套接字，添加到一个 `epoll` 实例中，准备传给 `listen_thread` 。在该 `epoll` 实例中，我们监听了它的读事件，以

及错误事件 `EPOLLERR`

```
{ /* Register listen fd to the listen_epfd */
    struct epoll_event event;
    event.data.fd = file_dsption;
    event.events = EPOLLET | EPOLLERR | EPOLLIN;
    epoll_ctl(listen_epfd, EPOLL_CTL_ADD, file_dsption, &e
vent);
}
```

- 紧接着，我们需要创建线程，用来完成接受创建新连接， 分配新连接， 处理新连接
- 先说前两个
- `listen_thread`

```
/* Listener's Thread
* @param arg will be a epoll instance
* */
static void * listen_thread(void * arg) {
    int listen_epfd = (int)arg;
    struct epoll_event new_client = {0};
    /* Adding new Client Sock to the Workers' thread */
    int balance_index = 0;
    while (terminal_server != CLOSE_SERVE) {
```

这是一个永不停止的循环，除非在外部传入了一个信号 `CTRL+C` ，其实没什么意义，不过还是写了

```
            //这是监听的阻塞地点，在此处会返回有多少个事件发生了，当然这里只有一个
            int is_work = epoll_wait(listen_epfd, &new_client, 1, 2000);
            int sock = 0;
            // 如果不是因为超时才到了这里
            while (is_work > 0) { /* New Connect */
                //接受并创建新连接
                sock = accept(new_client.data.fd, NULL, NULL);
                if (sock > 0) {
                    // 如果没有意外的话
                    set_nonblock(sock);
                    clear_clients(&clients[sock]);
                    clients[sock].file_dsp = sock;
                    // 分配新连接给各个workers_thread
                    add_event(epfd_group[balance_index], sock, EPOLLIN);
                    balance_index = (balance_index+1) % workers;
                } else /* sock == -1 means nothing to accept */
                    break;
            } /* new Connect */
        }/* main while */
        close(listen_epfd);
        pthread_exit(0);
    }
```

> 其实在上面的 `accept` 和 `set_nonblock` 可以用一个系统调用来解决， `accept4` ，而不需要使用两个不同的系统调用来完成这个功能，具体可以查询文档。

- 可以看出，这个 `listen_thread` 的职责非常简单，就只是单纯的接受创建新连接，设置一些属性，并且分配给 `workers_thread` ，所以真正复杂的工作还是在后者身上

workers_thread

- 这是整个程序的核心部分，但还是按照庖丁解牛的方法，一步步分解

- 整个的代码有点冗长，但是逻辑十分清晰，大体可以分成读写两部分

```
static void * workers_thread(void * arg) {
    int deal_epfd = (int)arg;
    struct epoll_event new_apply = {0};
    while(terminal_server != CLOSE_SERVE) {
        int is_apply = epoll_wait(deal_epfd, &new_apply, 1
, 2000);
        if(is_apply > 0) { /* New Apply */
            int sock = new_apply.data.fd;
            conn_client * new_client = &clients[sock];
```

到此处为止，前面的逻辑和 `listen_thread` 十分相似，需要额外说的就是 `epoll_wait` 接口中的第二，三个参数 ，代表着有事件改变状态的新连接(new_apply[i]),和有多少个这样的新连接(i)。代码中写的是 `(,&new_apply,1,)` 代表着我每次只想得到一个，说明及替代方案在后面会提到，跳过也无所谓。

```
            /* 读事件 */
            if (new_apply.events & EPOLLIN) { /* Reading W
ork */
                /* handle_read 是接收并解析HTTP请求报文的地方 *
/
                int err_code = handle_read(new_client);
                /* 此处省略一个很重要的分片错误处理 */
                else if (err_code != HANDLE_READ_SUCCESS)
{
                    /* Read Bad Things */
                    close(sock);
                    continue;
                }
            } // Read Event
```

以上便是简化的读事件的处理，抛开来看，一切的核心就是 `handle_read` 这个函数，后放会详细讲解。

```
                /* 写事件 */
                else if (new_apply.events & EPOLLOUT) { /* Wri
ting Work */
                    int err_code = handle_write(new_client);
                    /* TCP's Write buffer is Busy */
                    if (HANDLE_WRITE_AGAIN == err_code)
                        mod_event(deal_epfd, sock, EPOLLONESHO
T | EPOLLOUT);
                    else if (HANDLE_WRITE_FAILURE == err_code)
 { /* Peer Close */
                        close(sock);
                        continue;
                    }
                    /* if Keep-alive */
                    if(1 == new_client->conn_res.conn_linger)
                        mod_event(deal_epfd, sock, EPOLLIN);
                    else{
                        close(sock);
                        continue;
                    }
                } /* EPOLLOUT */
```

所谓 `clear_clients` 其实就是清除一些现有状态，不然下次有别的连接占用的时候就会错乱了。

```
                else { /* EPOLLRDHUG EPOLLERR EPOLLHUG */
                    close(sock);
                }
            } /* New Apply */
        } /* main while */
        return (void*)0;
    }
```

- 看起来有点长，实际上模块十分清楚。从上往下看，由三个 `if - else` 分支组成，分别处理 读事件，写事件，错误事件
- 这其中省略了一些十分重要的错误处理，以及某些优化，希望可以自己补全，但这都无所谓，因为已经将这种编程模型全盘托出，接下来就是细节方面的处理了。

handle_read

- 这应该是这个 HTTP服务器 真正的重点所在，用一个词来形容就是 核心技术，当然没那么高端，就是个程序而已。
- 前面提到一个名词，叫做 **HTTP**状态机，指的就是状态的转换，在C语言中，可以使用 `enum` 来实现

  ```
  typedef enum {
      HANDLE_READ_SUCCESS = -(1 << 1),
      HANDLE_READ_FAILURE = -(1 << 2),
      ...
  }HANDLE_STATUS;
  ```

  代表了， `handle_read` 是成功还是失败，有一个额外的 `MESSAGE_IMCOMPLETE` 状态也输一这个范畴内，但是设计的时候出现了差错，可以选择将其放在里面。

  `MESSAGE_IMCOMPLETE` 是为了应对**TCP**分片问题，所以在显示网络中很常见，但是本地测试的时候可能不容易发现，可以使用工具 `tc` 来模拟弱环境。

- `HANDLE_STATUS handle_read(conn_client * client)`

  ```
  HANDLE_STATUS handle_read(conn_client * client) {
      int err_code = 0;
      /* Reading From Socket */
      err_code = read_n(client);
      if (err_code != READ_SUCCESS) { /* If read Fail then E
  nd this connect */
          return HANDLE_READ_FAILURE;
      }
  ```

  到这里为止是读取所有可以读到的数据

```
        /* Parsing the Reading Data */
        err_code = parse_reading(client);
        if (err_code == MESSAGE_INCOMPLETE)
            return MESSAGE_INCOMPLETE;
        if (err_code != PARSE_SUCCESS) { /* If Parse Fail then
End this connect */
            return HANDLE_READ_FAILURE;
        }
```

到这里为止是处理所有已经读到的数据

```
        return HANDLE_READ_SUCCESS;
    }
```

到了这里，就证明读和处理都已经正确完成了。

> 巧用 `gdb` 能让你轻松理解整个状态机的逻辑

- 从函数接口上看，它接受一个 `conn_client` 类型的指针，回想一下，这就是我们存储每个新连接的各种信息的地方，返回值就是这个动作的状态了。
- 从功能上看，这个函数主要的工作就是将 `handle_read` 拆分成两大部分：

  1. 读取数据 (`read_n`)
     i. 首先读取所有能读取的数据（从 `socket` 中）
     ii. 验证数据是否完整
        i. 对于 `GET` 而言就是是否读取到了一个空行 `\r\n`
        ii. 对于 `POST` 来说就是是否一句 `Content-length` 属性的值将 **body** 读取完整了
  2. 处理数据 (`parse_reading`)
     i. 处理**HTTP**请求报文第一行状态行
     ii. 处理剩余的头属性，如 `Connection`
     iii. 生成响应报文，你可以考虑将这一步划分出去，因为这一步涉及到了磁盘**I/O**

- 先说第一部分，读取数据**(`read_n`)**
- `static int read_n(conn_client * client)`
- 实现一个 `read` 函数的加强版

```
__thread char read_buf2[CONN_BUF_SIZE] = {0};
static int read_n(conn_client * client) {
    int    read_offset2 = 0;
    int    fd         = client->file_dsp;
    char * buf        = &read_buf2[0];
    int    buf_index = read_offset2;
    int read_number = 0;
    int less_capacity = 0;
```

从前往后依次是读缓冲区位移， 处理的连接套接字, `buf` 纯粹多此一举还可能阻碍编译器优化，但我还是写了，强迫症吧， `buf_index` 同理， `read_number` 是本次读的字符个数， `less_capacity` 是缓冲区的容量余量

```
    while (1) {
        /* 因为是非阻塞，所以要不停地读，直到`read`返回-1，且errn
o为EAGAIN */
        less_capacity = CONN_BUF_SIZE - buf_index;
        if (less_capacity <= 1) {/* Overflow Protection */
            /* 万一这本地的缓冲区容量不够了，就刷新进 conn_client
 中 */
            buf[buf_index] = '\0'; /* Flush the buf to the
 r_buf String */
            /* 对于 STRING 宏，可以看看我的源码中的 wsx_string.
h */
            cappend_string(client->r_buf, STRING(buf));
            client->r_buf_offset += read_offset2;//- clien
t->read_offset;
            read_offset2 = 0;
            buf_index = 0;
            less_capacity = CONN_BUF_SIZE - buf_index;
            /* 清空缓冲区成功 */
        }
```

上面的代码中，有一个 `APPEND` 宏，是用来简化代码的，功能是 `#define APPEND(str) str,(strlen(str)+1)`

```
          read_number = (int)read(fd, buf+buf_index, less_ca
pacity);
          /* 0代表对端关闭了连接或者说是已经读完了 EOF(对端调用close
()/shutdown()) */
          if (0 == read_number) { /* We must close connectio
n */
              return READ_FAIL;
          }
          /* -1 代表现在没东西可以读了 */
          else if (-1 == read_number) { /* Nothing to read *
/
              if (EAGAIN == errno || EWOULDBLOCK == errno) {
                  /* 这个时候，我们该做的就是将缓冲区的东西，存储起
来 */
                  buf[buf_index] = '\0';
                  append_string(client->r_buf, STRING(buf));
                  client->r_buf_offset += read_offset2;//cli
ent->read_offset;
                  return READ_SUCCESS;
              }
              return READ_FAIL;
          }
          else { /* Continue to Read */
              /* 能读取到信息，就继续读 */
              buf_index += read_number;
              read_offset2 = buf_index;
          }
      } /* while(1) */
  }
```

> `__thread` 关键字是多线程编程里一个挺有用的一个关键字，具体可以查询资料，简单来说，就是让每个线程拥有一个自己的全局变量。

- 经过 `read_n` 之后，我们就(可能)获取到了完整的数据了，接下来就是解析它们，引入一个状态
- `PARSE_STATUS`

```
typedef enum {
/* Parse the Reading Success, set the event to Write Event
*/
    PARSE_SUCCESS    = 1 << 1,
/* Parse the Reading Fail, for the Wrong Syntax */
    PARSE_BAD_SYNTAX = 1 << 2,
/* Parse the Reading Success, but Not Implement OR No Such
Resources*/
    PARSE_BAD_REQUT  = 1 << 3,
}PARSE_STATUS;
```

解释的很清楚了，不再赘述。

- `PARSE_STATUS parse_reading(conn_client * client)`

```
PARSE_STATUS parse_reading(conn_client * client) {
    int err_code = 0;
    requ_line line_status = {0};
    client->r_buf_offset = 0; /* Set the real Storage offs
et to 0, the end of buf is '\0' */
```

`requ_line` 是一个结构体，用来存储状态行所含有的三个信息: 请求方法，请求资源，**HTTP**版本号

```
    /* Get Request line */
    err_code = deal_requ(client, &line_status);
    /* 回想一下这个状态，TCP分片的情况 */
    if (MESSAGE_INCOMPLETE == err_code)  /* Incompletely r
eading */
        return MESSAGE_INCOMPLETE;
    if (DEAL_LINE_REQU_FAIL == err_code) /* Bad Request */
        return PARSE_BAD_REQUT;
```

到这里为止是处理状态行的代码

```
        /* Get Request Head Attribute until /r/n */
        err_code = deal_head(client);  /* The second line to t
he Empty line */
        if (DEAL_HEAD_FAIL == err_code)
            return PARSE_BAD_SYNTAX;
```

到这里为止是处理完了所有的头属性

```
        /* Response Page maker */
        err_code = make_response_page(client);
        if (MAKE_PAGE_FAIL == err_code)
            return PARSE_BAD_REQUT;

        return PARSE_SUCCESS;
    }
```

- 对于 `deal_requu` ， `deal_head` 来说，只是一个很简单的从大字符串中识别出小字符串，并存储起来的问题，不想过多的叙述。在这个处理过程中，自己实现了一个 `get_line` 按行读取的函数，同样会被后面的 `deal_head` 使用
    - 这其中有一些问题需要注意一下，那就是你需要考虑**TCP**分片问题，这是我第三次提到这个东西，也就是用状态机监测好这个问题是否发生，并及时处理。
    - 在 `deal_head` 中，可以按行进行循环读取(`get_line`)，知道你发现空行，那么你就处理完成了，如果是 `POST` 方法，你还需要继续读取，直到读取完它的**body**。现在想想， `conn_client` 这个结构体中的那些属性是干什么的，就是从这里解析出来的。
- 读取解析完成之后，就能进行响应报文的生成了。在下一节中详述

题外话

- 和上一个部分不同，再上一个部分我尽可能的不落下一丝一毫的细节，将自己如何写程序的想法分享给诸位
- 但这章节，无论怎么看，从思维，从代码都不再像之前那般面面俱到，我认为也没有必要，这一章大家应该就具备了自我独立思考的能力，实际上在给出了结构图之后，后面的章节就不怎么必要了
- 但我想把自己的想法写出来，想想求学的这几年无人引导，苦苦寻找资料的那

些日子，我觉得我有必要把自己从网络上得来的知识，再次回馈给网络，这才是生生不息，自我进步的道理。

最后

- 额外的补充
    - 我的博客提到了一些错误处理的详细解释：一个HTTP服务器的C之路(上).html)
    - 陈硕的书 ：《Linux多线程服务端编程》
- 这个经验分享系列马上就要到头了，下一步的我也许就该毕业了
    - 也许在最后一年，我会用最后的时间完成额外的章节
    - 额外的章节有过想法，就是写一个完整可用的数据库系统
    - 这个工程量远超前方章节，如果有想法我会及时在本书中更新动态
    - 希望大家也能够将自己知道的，学到的知识贡献出来
- 如果觉得我说的还行，可以给我来一点鼓励呀
    -

下一节

- 讲述如何生成响应报文，以及本章的收尾。

0x17-套接字编程-HTTP服务器(5)

- 让我们停下来，回想一下之前的内容

    1. 首先读取配置文件，并凭此打开服务器套接字
    2. 确定一切完备的情况下(`listen`)，开启事务循环 `handle_loop`
    3. 准备好各项资源 `prepare_worker` ，开启两种线程就真正开始工作了

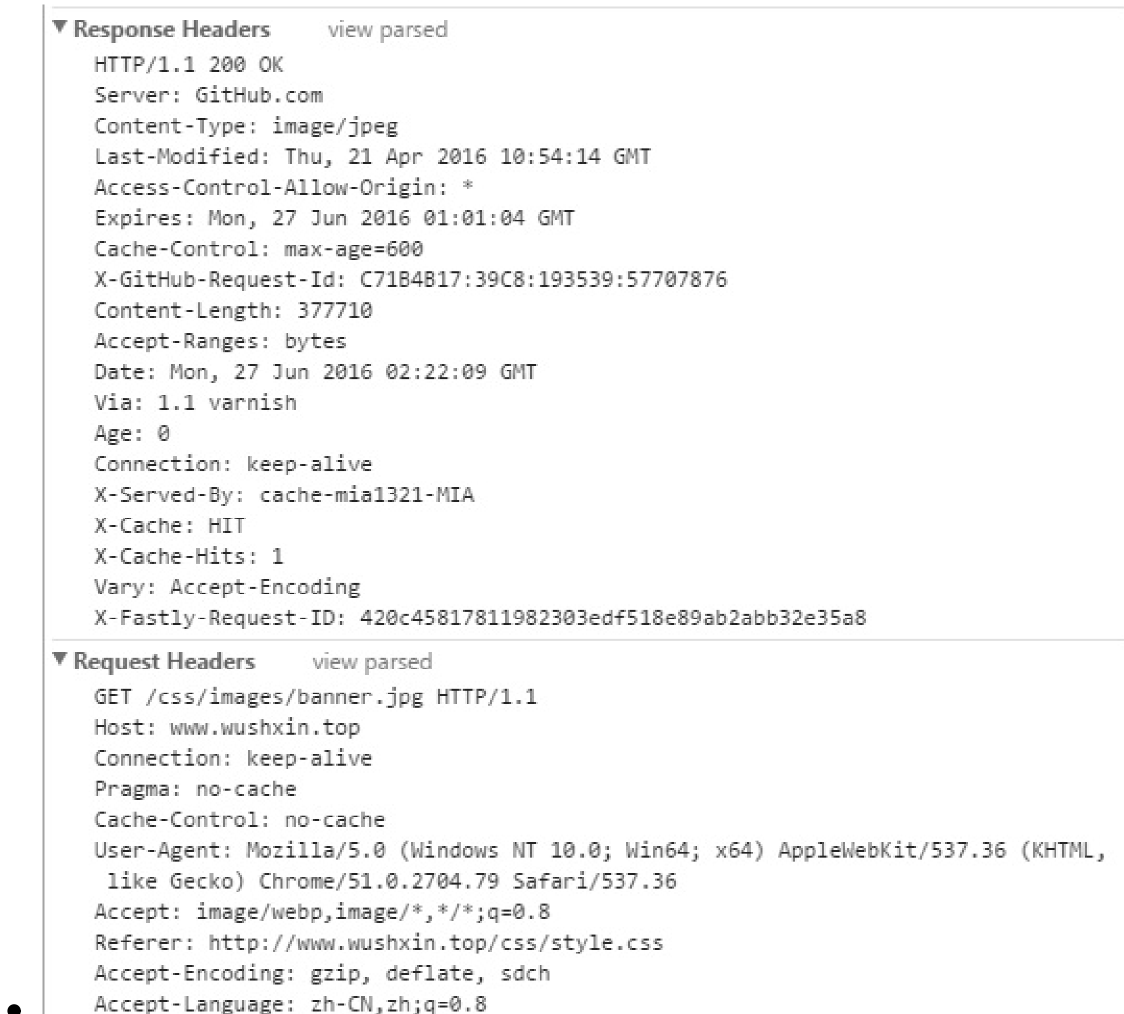
- `string_t` 不打算详细讲解，因为并不是什么好的设计，但是只需要将接口，改成C风格的就不错，但是有一个致命的缺点，就是这不是二进制字符串

    - 什么意思？就是这是一个C风格的字符串，无法很好的存储二进制数据，例如无法存储 `\0` 这个字符，实际上要设计就需要重新设计。
    - 但这个小程序绰绰有余，因为只是作为一个静态资源HTTP服务器在使用。
    - 在本章最后，会将源代码地址贴上，仅供参考，写的不够严谨，但还是有意义的练习。

> 2016-08-28 修复上述问题，具体可以参看源码，现在支持二进制数据
>
> 万事开头难，当你在键盘上打下第一句代码的时候你就成功了。看永远都只能是谈谈兵，虽说谈兵也需要技术

生成一个响应报文

- 实际上客户端对你怎么处理这些数据一点都不感兴趣，他们感兴趣的不就是你的响应报文是什么吗
- 所以说到了这一步就要看看这个报文的组成，但这并不是我们的重点，简单讲一下哪些属性比较重要。
- 还记得开头的时候，给出了一个报文实例，实际上最明了的莫过于在浏览器中摁 `F12` 后自己查看交互报文，再专业一些使用 `Wireshark` 这类专业抓包软件也未尝不可，以浏览器为例：

- 

    - 这是个人博客上的一个背景图的请求交互，重点看 `Response Headers`
    - 这么一长串，实际上真正必不可少的还是那么两行
        - `HTTP/1.1 200 OK`
        - `Content-Length: 377710`
    - 前者告诉你这个球球的结果，后者告诉你请求的结果的内容在哪里，即在报文中空行后多少个字节都是请求的结果。
- 那在C语言中，或者说在任何语言中，都没什么特别好的办法，就是用字符串构造报文了。作为一个标准库比较贫瘠的语言，这就要我们多做一点工作，这也是为什么要自己写一个字符串结构体的原因所在。
    - 当然如果你为了兼容二进制数据，那么甚至连标准库中的字符串函数都不能使用了，包括 `Linux` 提供的扩展 `gnu99` 字符串函数，原因是因为C-Style字符串是以 `\0` 作为结束符的。
- 现在我们规定一下，我们这个服务器的响应报文会包含的部分

    1. 状态行是必要的 `HTTP/VER STATUS_CODE STATUS_MESSAGE\r\n`
    2. 服务器时间 `Date: xxx\r\n`
        - 用的是UTC格式，实际上此处也可以有点小讲究，后面提一下

3. 资源类型 `Content-Type: xxx\r\n`
4. 资源长度 `Content-Length: xxx\r\`
5. 连接状态 `Connection: xxx\r\n`
6. 空行 `\r\n`
7. 资源

- 在进入生成报文的环节中，其实还有很多工作要做，例如判断是什么请求方法，是否是恶意请求， 获取资源的各种信息 等，直接进入最核心的阶段 `make_response` 中的 `write_to_buf`
- 也就是构造报文阶段

```
  __thread char local_write_buf[CONN_BUF_SIZE] = {0};
  static int write_to_buf(conn_client * restrict client, //
connection client message
                  const char * const * restrict status,  int
 rsource_size) {
  #define STATUS_CODE 0
  #define STATUS_TITLE 1
  #define STATUS_CONTENT 2
      char *   write_buf = &local_write_buf[0]; /* Local wri
te buffer */
      string_t resource  = client->conn_res.requ_res_path; /
* Resource that peer request */
      string_t w_buf     = client->w_buf;     /* Real data b
uffer */
      int w_count = 0;
      struct tm * utc;   /* Get GMT time Format */
      time_t      now;
      time(&now);
      utc = gmtime(&now);/* Same As before */
```

`utc` 此时并不是标准的格式字符串，但这个变量里面有我们需要的资源

```
        /* Construct the HTTP head */
        w_count += snprintf(write_buf+w_count, CONN_BUF_SIZE-w
_count, "%s %s %s\r\n",
                        http_ver[client->conn_res.request_http_v],
                        status[STATUS_CODE], status[STATUS_TITLE])
;
        w_count += snprintf(write_buf+w_count, CONN_BUF_SIZE-w
_count, "Date: %s, %02d %s %d %02d:%02d:%02d GMT\r\n",
                        date_week[utc->tm_wday], utc->tm_mday,
                        date_month[utc->tm_mon], 1900+utc->tm_year
,
                        utc->tm_hour, utc->tm_min, utc->tm_sec);
        w_count += snprintf(write_buf+w_count, CONN_BUF_SIZE-w
_count, "Content-Type: %s\r\n", content_type[client->conn_re
s.content_type]);
        w_count += snprintf(write_buf+w_count, CONN_BUF_SIZE-w
_count, "Content-Length: %u\r\n", 0 == rsource_size

                                ? (unsigned int)strlen(status[2]):
(unsigned int)rsource_size);
        w_count += snprintf(write_buf+w_count, CONN_BUF_SIZE-w
_count, "Connection: close\r\n");
        w_count += snprintf(write_buf+w_count, CONN_BUF_SIZE-w
_count, "\r\n");
        write_buf[w_count] = '\0';
```

从上往下依次是刚才我在上面介绍的顺序，使用的是 `snprintf` 函数，其实此处可以将这些语句合并起来写，而不是分别调用，十分浪费。但这么写比较清晰

其中在生成时间的时候，我使用的是预定义好的静态字符串数组来帮助我，可很好的猜到这些 `date_xxx` 数组里放的都是些什么，无非就是一些时间的缩写。

```
        /* 写入缓冲区 */
        append_string(w_buf, STRING(write_buf));
        client->w_buf_offset = w_count;

        /* If Server do not wanna to sent local file */
        if (0 == rsource_size) {  /* GET Method */
            append_string(w_buf, STRING(status[STATUS_CONTENT]
));
            snprintf(write_buf+w_count, CONN_BUF_SIZE-w_count,
 status[2]);
            return 0;
        } else if (-1 == rsource_size) { /* HEAD Method */
            return 0;
        }
        /* 如果需要服务器上的实体资源，那就找到它 */
        int fd = open(resource->str, O_RDONLY);
        if (fd < 0) {
            return -1; /* Write again */
        }
        /* 将资源文件映射到内存里，这样就能很好的操作 */
        char *file_map = mmap(NULL, (size_t)rsource_size, PROT
_READ, MAP_PRIVATE, fd, 0);
        if (NULL == file_map) {
            assert(file_map != NULL);
    }
        close(fd);
        /* 存入缓冲区 */
        append_string(w_buf, file_map, rsource_size);
        client->w_buf_offset += rsource_size;
        munmap(file_map, (unsigned int)rsource_size);
        return 0;
    }
```

- 上面有几个函数调用 `open` , `mmap` , `munmap` ，学过 `Linux` 系统编程的人肯定知道，这是共享内存的一种最简单高效的方式。
- 看不太懂的可以去查询 `APUE` 或者网上资源很多，这是很重要的一个知识点。大致的功能就是将一个文件打开，并映射到内存中，这个内存可以在多个进程间共享 `MAP_SHARED` 也可以不共享 `MAP_PRIVATE` ，这样我们就能像数组一样

对其进行读取操作了。

- 至于 `make_response_page` 的代码就不贴源码了，因为代码几乎都是在做检测的工作，例如安全之类的事情，以及方法分配，只需要扫一眼就能够很清楚的理解了。
- 在构造完成报文之后，下一步自然就是发送它了，那我们又回到了 `worker_thread` 中去

发送报文

- 那这个就简单很多了，直接贴上代码

```
HANDLE_STATUS handle_write(conn_client * client) {
    /* String Version */
    char*    w_buf    = client->w_buf->str;
    int      w_offset = client->w_buf_offset;
    int nbyte = w_offset;
    int count = 0;
    int fd = client->file_dsp;
    while (nbyte > 0) {
        w_buf += count;
        count = write(fd, w_buf, nbyte);
        if (count < 0) {
            if (EAGAIN == errno || EWOULDBLOCK == errno) {
                /* 如果发送缓冲区不够容纳所有的，那就下次再发 */
                memcpy(client->w_buf->str, w_buf, strlen(w
_buf));
                client->w_buf_offset = nbyte;
                return HANDLE_WRITE_AGAIN;
            }
            /* 在这个地方就是前面所说的那个EPIPE错误 */
            else /* if (EPIPE == errno) */
                /* 对端关闭了连接 */
                return HANDLE_WRITE_FAILURE;
        }
        else if (0 == count)
            return HANDLE_WRITE_FAILURE;
        nbyte -= count;
    }
    return HANDLE_WRITE_SUCCESS;
}
```

- 就是这么简单，因为实在是没有其他工作可以做了
    - 尝试发送所有，直到发送完全部数据，或者发送缓冲区不够，那就等待下次发送，这个通过 `epoll` 很容易就实现了。
    - 如果发现对面的不在了，直接关闭就好啦。

附加

- 源码地址

- 个人博客

小结

- 其实也是拖拖拉拉地在不断地写这些东西
- 也还是因为时间不多的原因，一直想抽一个连贯的时间，结果一拖就是半年，所以做事一定要当机立断，当然要经过脑子。看起来挺矛盾
- 写到这里，算是给自己的求学之路一个挺好的交代，因为至少将自己知道的都写了出来，对我也好，对其他人也好，至少挺安心的。
- 无论如何都要感谢一下互联网，学校图书馆的馆藏和荐购权限。
- 不知道我这些东西有多少能帮助到看的人，但我知道一定会有影响，也一定有不好的地方，但是我不怕，就怕没人和我说我错在哪里。
- 接下来我想做的事就是用剩下的一年里去互联网，IT的各个大领域实习，见见世面，心中还是有鸿鹄之志的。
- 这本书也就到此为止了

题外话

- 实际上也是构思了三个月左右，我打算附加一章，用来实现一个数据库系统，在上一节也提到过。
- 大致的想法是实现 ： SQL编译器，数据库存储引擎，数据库管理系统。至于事务的话，看看吧。觉得如果我写下来就一定会和大家分享。谢谢给我支持的那些人。